In [1]:
```
%matplotlib inline

from matplotlib import pyplot as plt
import numpy as np
import imp
from IPython.display import YouTubeVideo
from IPython.display import HTML
```

In [2]:
```
from PIL import Image, ImageChops

def trim(im, percent=36):
    bg = Image.new(im.mode, im.size, im.getpixel((0,0)))
    diff = ImageChops.difference(im, bg)
    diff = ImageChops.add(diff, diff, 2.0, -100)
    bbox = diff.getbbox()
    if bbox:
        x = im.crop(bbox)
        return x.resize(((x.size[0]*percent)/100, (x.size[1]*percen
t)/100), Image.ANTIALIAS)

def resize(filename, percent=36):
    trim(Image.open(filename + ".png"), percent).save(filename + "_
r" + str(percent) + ".png")
```

EECS 545: Machine Learning

Lecture 22: Neural Networks (Part 1)

- Instructor: **Junhyuk Oh**
- Date: April 6, 2016

Outline

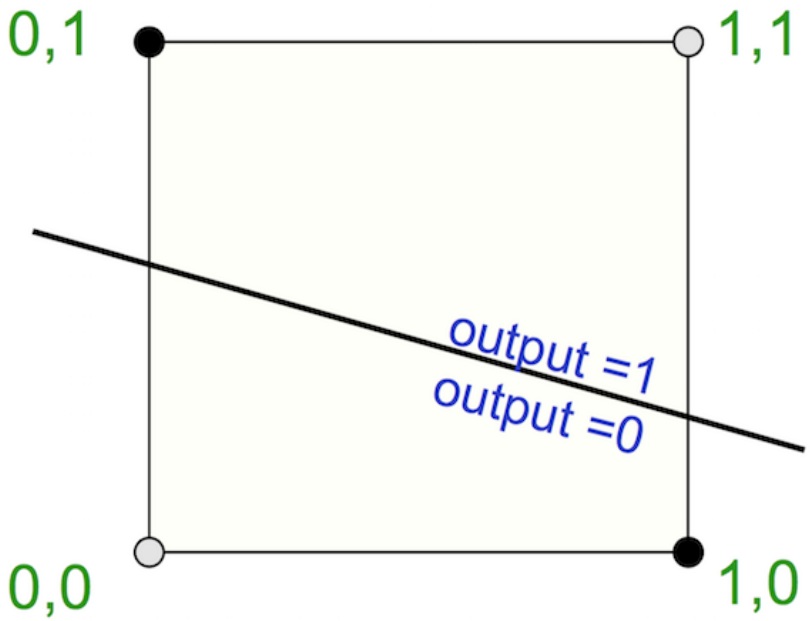
- Motivation
- Basics of Neural Networks
  - Forward Propagation
  - Backward Propagation
- Deep Neural Networks
  - Convolutional Neural Networks
  - Recurrent Neural Networks
- Applications
  - Computer Vision
  - Natural Language Processing
  - Reinforcement Learning

Limitations of Linear Classifiers

- Linear classifiers (e.g., logistic regression) classify inputs based on linear combinations of input $\boldsymbol{x_i}$
- Many decisions involve non-linear functions of the input

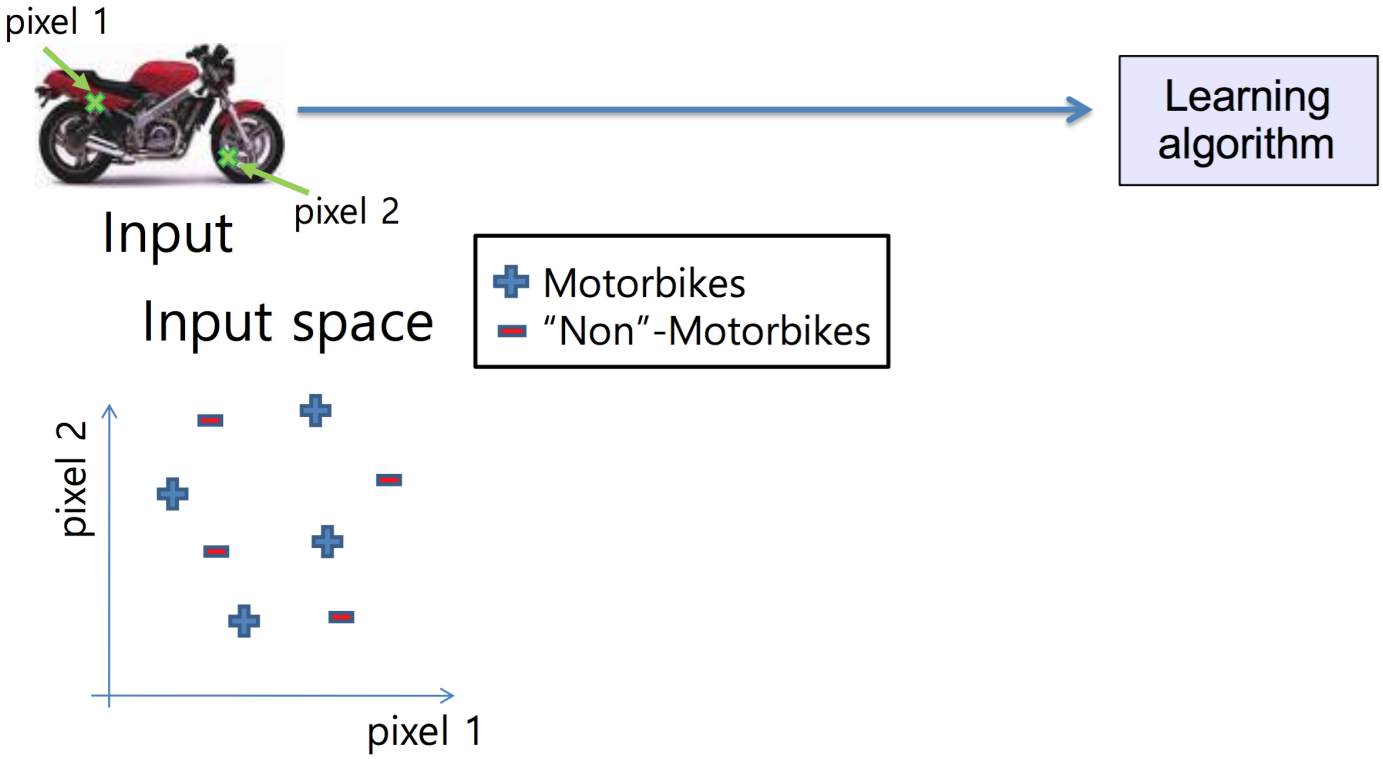
Limitations of Linear Classifiers

- Canonical example (XOR function)
  - The positive/negative examples are not *linearly separable*.
  - Need to map the input (x_1, x_2) to a feature space where examples are linearly separable.



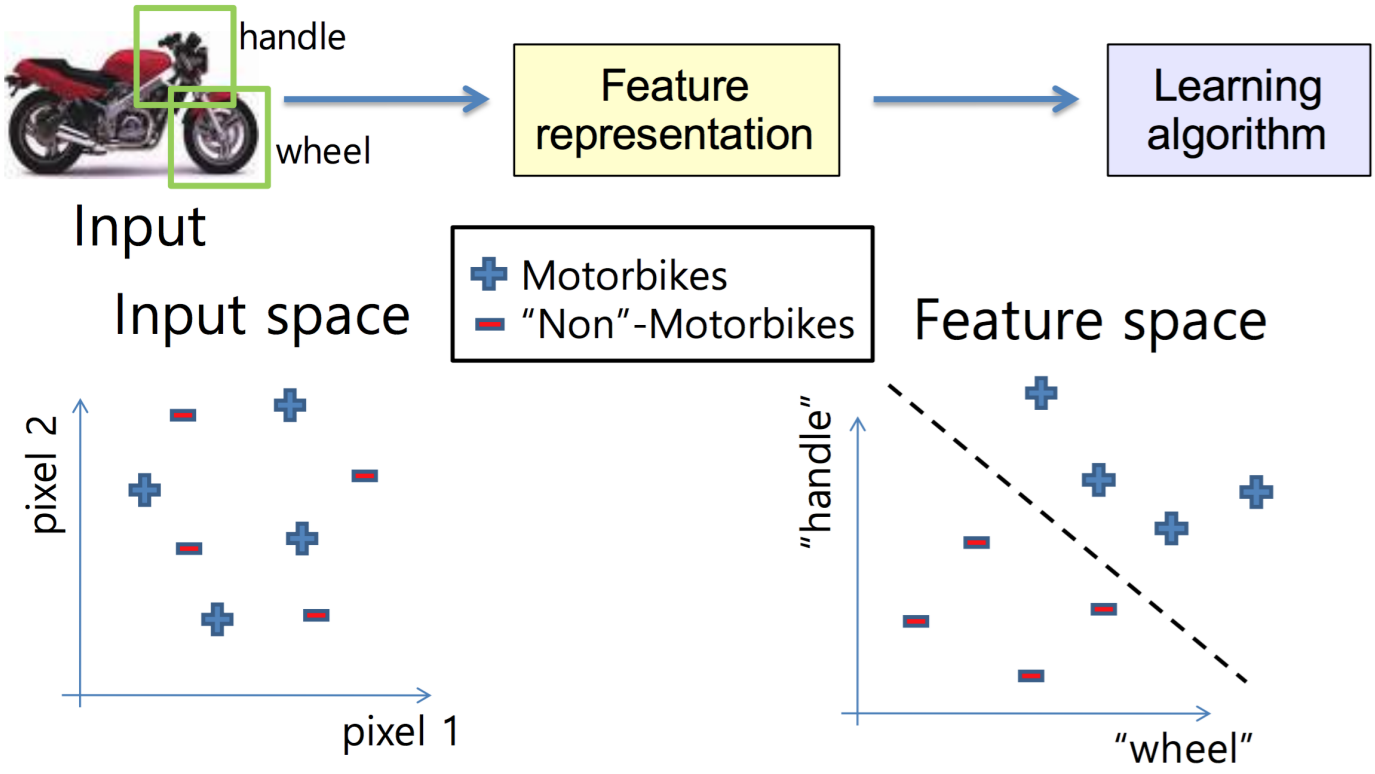
(Figure from Raquel Urtasun & Rich Zemel)

Feature Representation



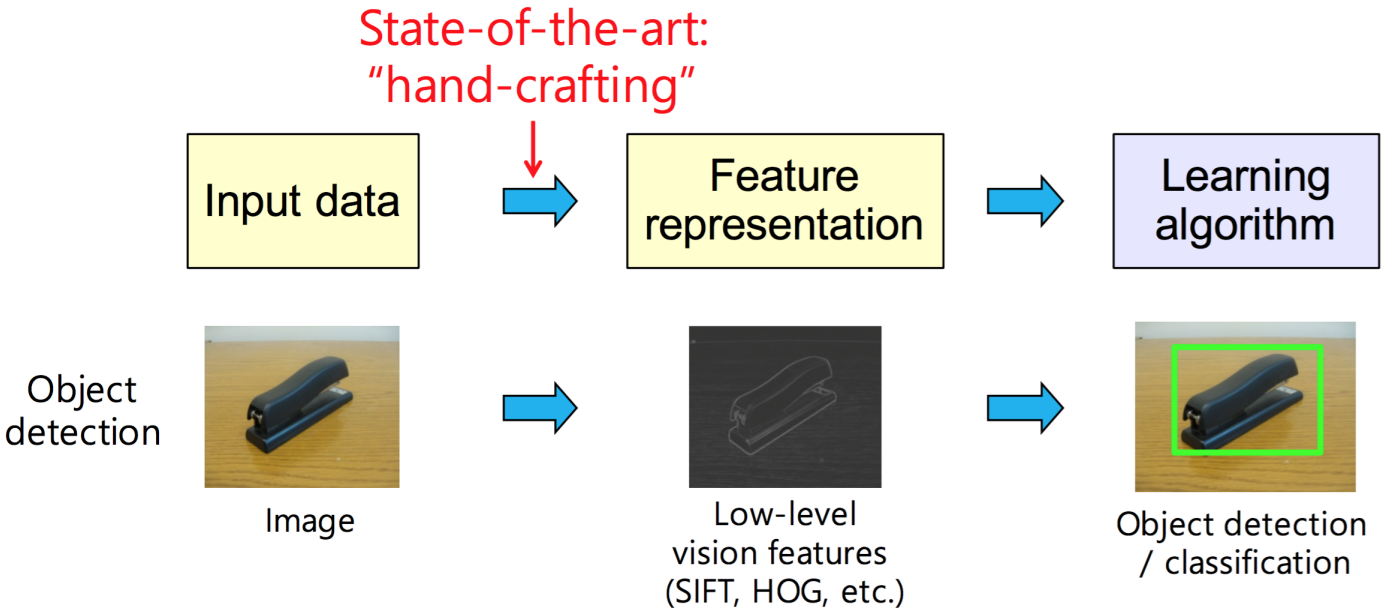
(Slide Credit: Honglak Lee)

Feature Representation



(Slide Credit: Honglak Lee)

Hand-crafted Feature Representation



(Slide Credit: Honglak Lee)

Drawbacks of Hand-crafted Feature Representation

- Requires expert knowledge
- Requires time-consuming hand-tuning

Q) Can we learn useful features from raw data?

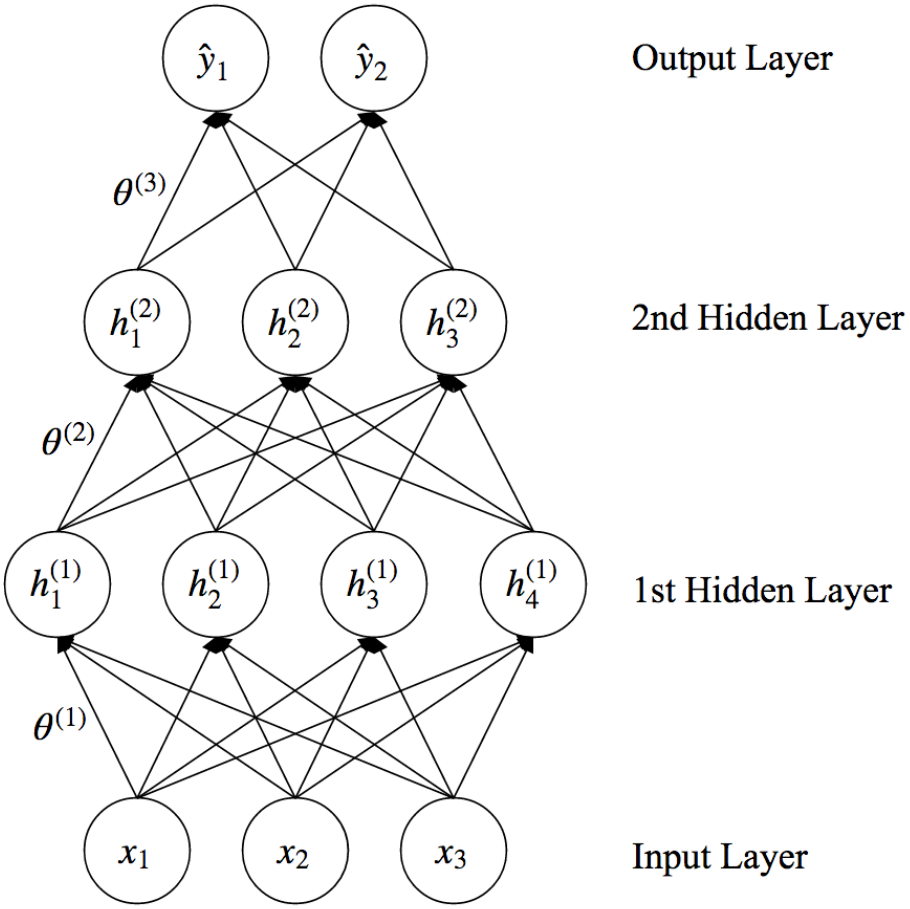
→ Yes. That's what **deep learning** is trying to do.
→ **Neural network** can implement the idea of deep learning.

Outline

- Motivation
- **Basics of Neural Networks**
  - Forward Propagation
  - Backward Propagation
- Deep Neural Networks
  - Convolutional Neural Networks
  - Recurrent Neural Networks
- Applications
  - Computer Vision
  - Natural Language Processing
  - Reinforcement Learning

Overview of Neural Networks

- Input Layer: provides input
- Hidden Layer: features extracted from input
- Output Layer: output of the network
- Parameters (or weights) for each layer



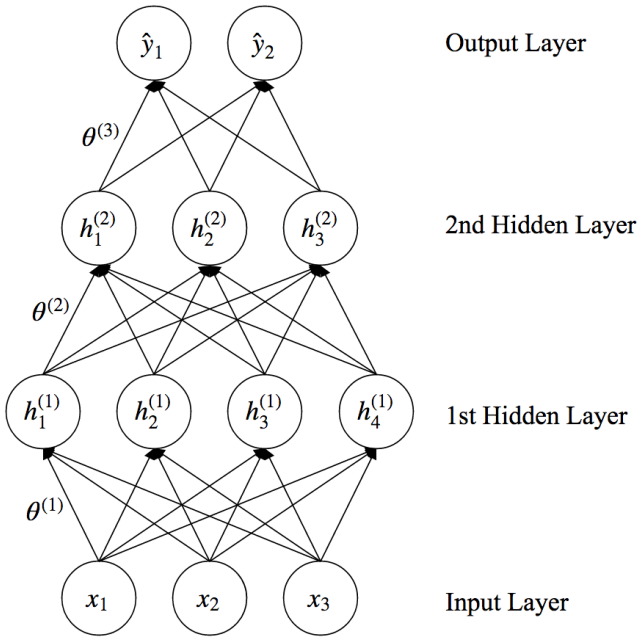
Overview of Neural Networks

- A **loss function** is defined over the *output units* and *desired outputs* (i.e., labels)

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) \text{ where } \hat{\mathbf{y}} = f(\mathbf{x}; \theta)$$

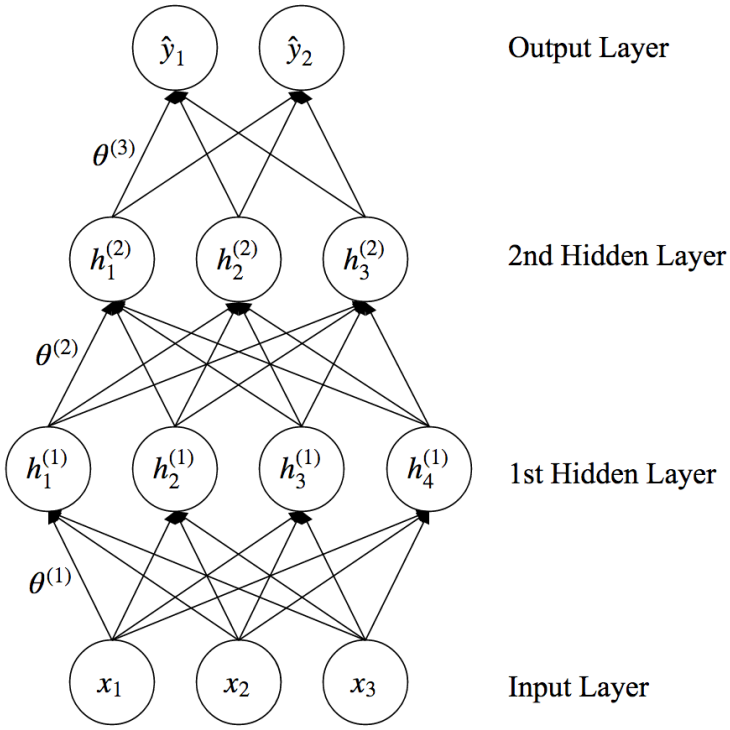
- The parameter of the network is trained to minimize the loss function based on gradient descent methods

$$\min_{\theta} \mathbb{E}_{(\mathbf{x},\mathbf{y}) \sim \text{ data}} [\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})] \text{ where } \hat{\mathbf{y}} = f(\mathbf{x}; \theta)$$



Overview of Neural Networks

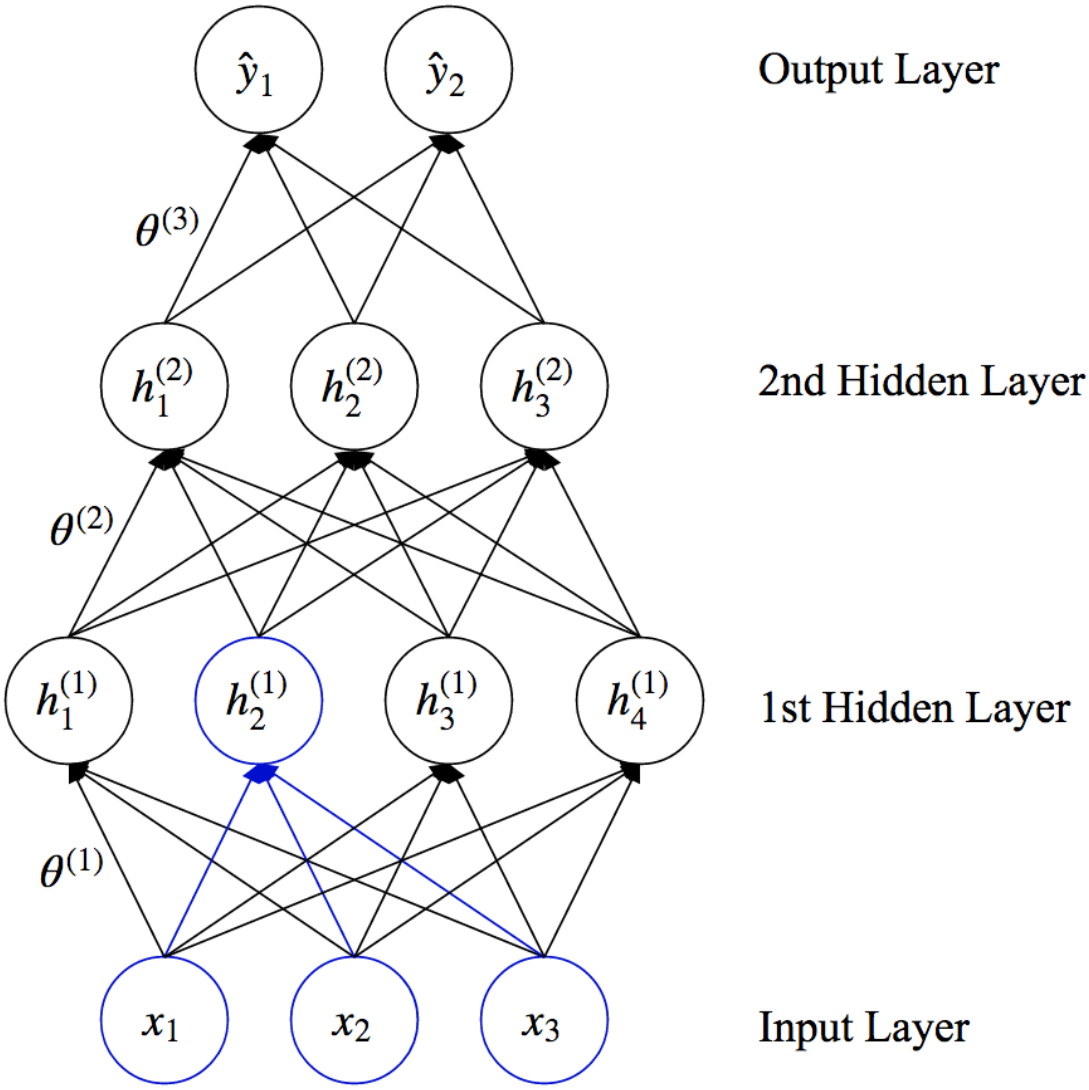
- Forward Propagation (inference): Compute $\hat{\mathbf{y}} = f(\mathbf{x}; \theta)$ (output given input)
- Backward Propagation (learning): Compute $\nabla_\theta \mathcal{L}$ (gradient of loss w.r.t. parameters)



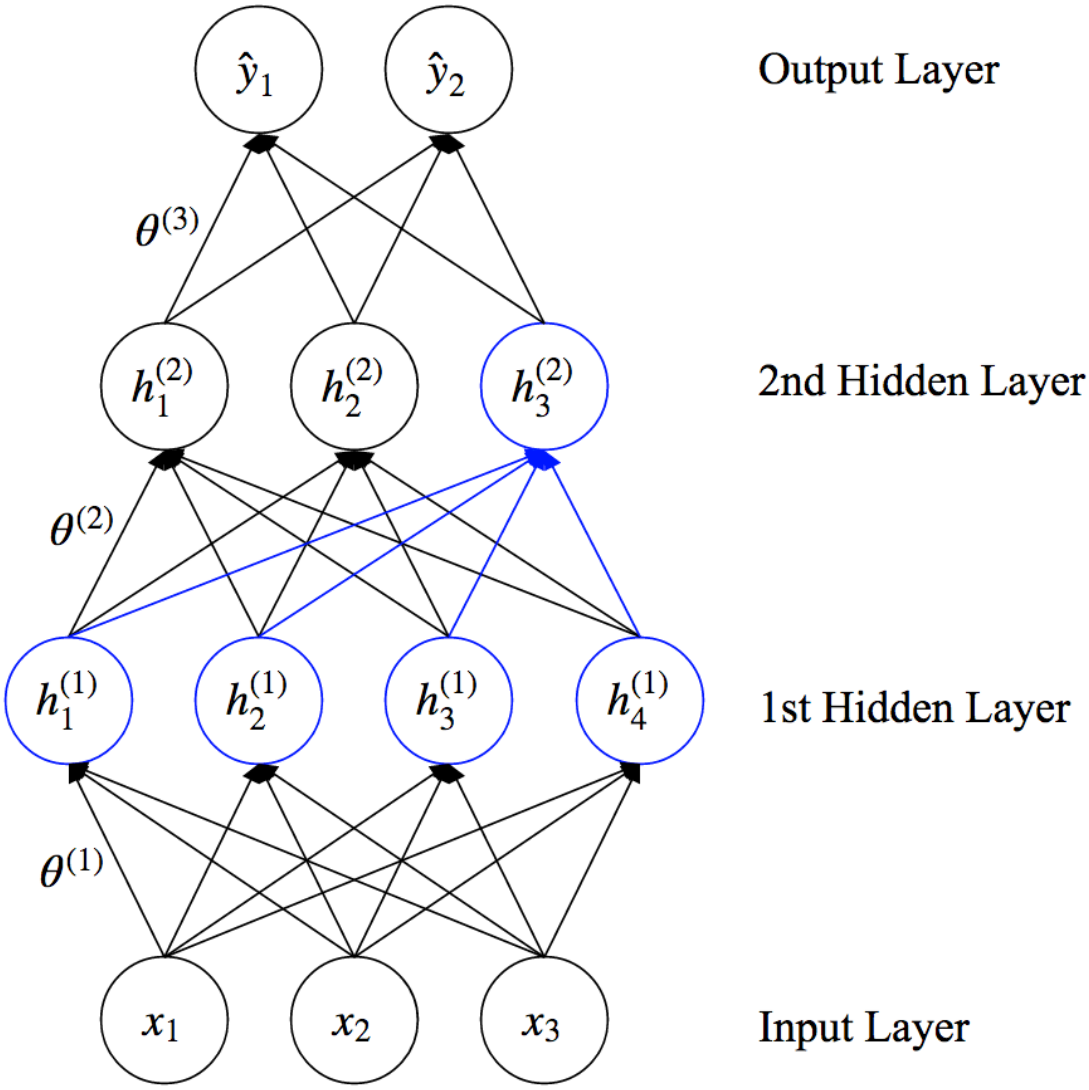
Outline

- Motivation
- Basics of Neural Networks
  - **Forward Propagation**
  - Backward Propagation
- Deep Neural Networks
  - Convolutional Neural Networks
  - Recurrent Neural Networks
- Applications
  - Computer Vision
  - Natural Language Processing
  - Reinforcement Learning

Forward Propagation



Forward Propagation

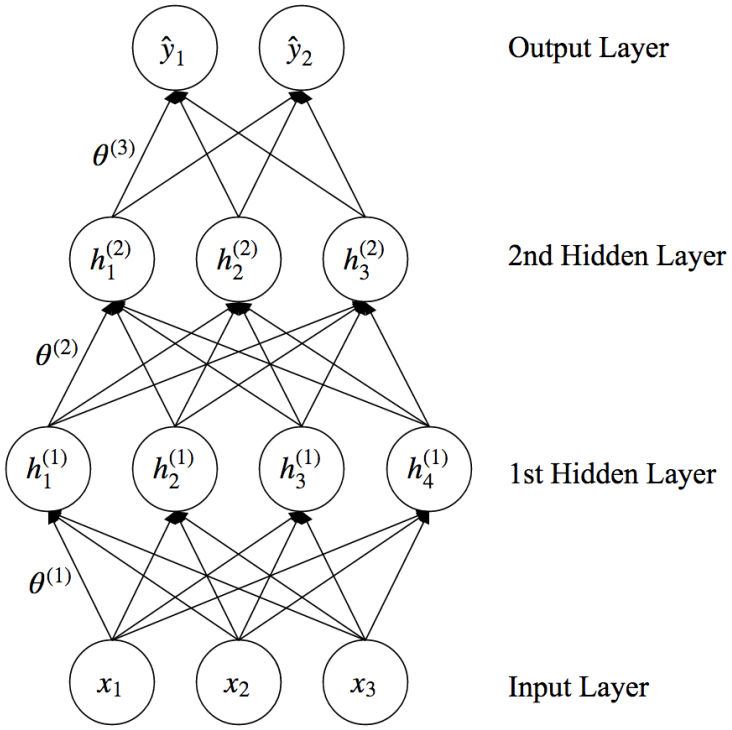


Forward Propagation

- The activation of each unit is computed based on **the previous layer** and **parameters (or weights)** associated with edges

$$\underbrace{\mathbf{h}^{(l)}}_{l\text{-th layer}} = f^{(l)}\Big(\underbrace{\mathbf{h}^{(l-1)}}_{(l-1)\text{-th layer}} ; \underbrace{\theta^{(l)}}_{\text{weights}} \Big) \text{ where } \mathbf{h}^{(0)} \equiv \mathbf{x}, \mathbf{h}^{(L)} \equiv \hat{\mathbf{y}}$$

$$\hat{\mathbf{y}} = f(\mathbf{x}; \theta) = f^{(L)} \circ f^{(L-1)} \cdots f^{(2)} \circ f^{(1)} \left(\mathbf{x}; \theta^{(1)}\right)$$

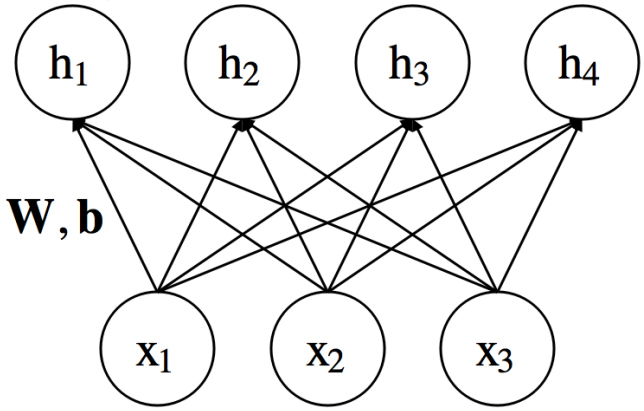


Types of Layers: Linear

$$h_i = \sum_j w_{ij} x_j + b_i$$

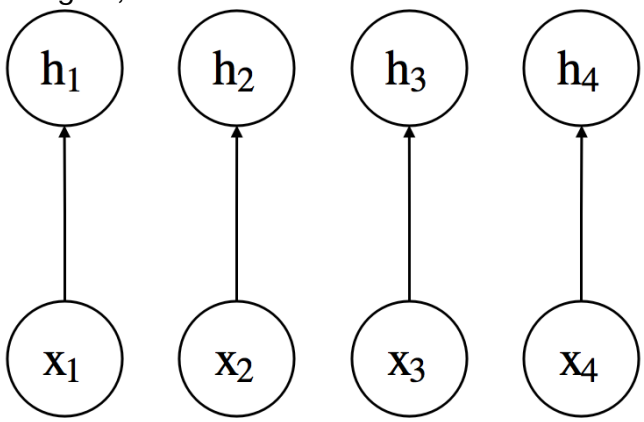
$$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

- $\mathbf{x} \in \mathbb{R}^m$: Input, $\mathbf{h} \in \mathbb{R}^n$: Output
- $\mathbf{W} \in \mathbb{R}^{n \times m}$: Weight, $\mathbf{b} \in \mathbb{R}^n$: Bias $\rightarrow$ parameter
- Often called "fully-connected layer"



Types of Layers: Non-linear Activation Function

- Applies a non-linear function to individual units.
- There is no weight.
- Allows neural networks to learn non-linear features.
- ex) Sigmoid, Hyperbolic Tangent, Rectified Linear Function



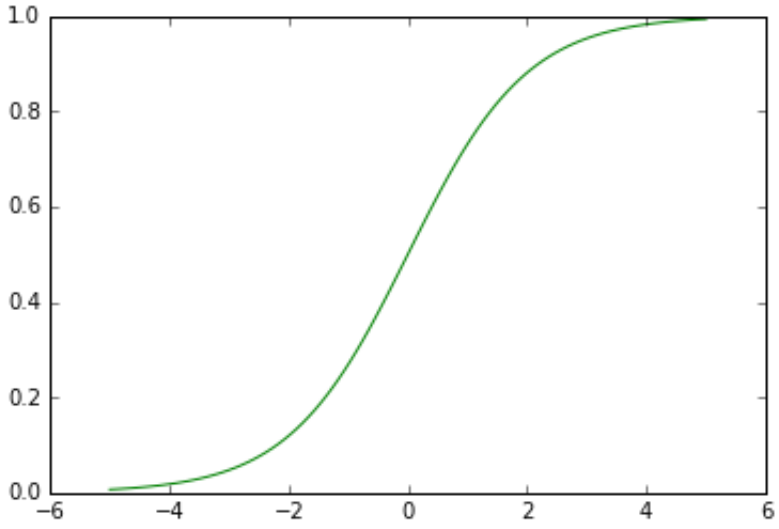
Non-linear Activation: Sigmoid

$$h_i = \sigma(x_i) = \frac{1}{1 + \exp(-x_i)}$$

$$\mathbf{h} = \sigma\left(\mathbf{x}\right)$$

In [3]:
```
xx = np.linspace(-5, 5, 100)
plt.plot(xx, 1/(1 + np.exp(-1 * xx)), '-g')
```

Out[3]: [<matplotlib.lines.Line2D at 0x7f25e0f35150>]



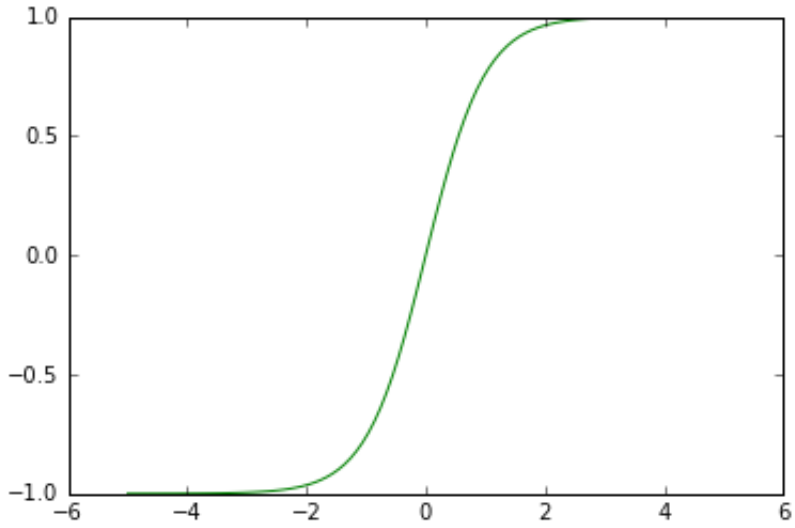
Non-linear Activation: Hyperbolic Tangent (Tanh)

$$h_i = \tanh(x_i) = \frac{\exp(x_i) - \exp(-x_i)}{\exp(x_i) + \exp(-x_i)}$$

$$\mathbf{h} = \tanh\left(\mathbf{x}\right)$$

In [4]:
```
xx = np.linspace(-5, 5, 100)
plt.plot(xx, (np.exp(xx) - np.exp(-1 * xx))/(np.exp(xx) + np.exp(-1
* xx)), '-g')
```

Out[4]:
```
[<matplotlib.lines.Line2D at 0x7f25de437190>]
```



Non-linear Activation: Rectified Linear (ReLU)

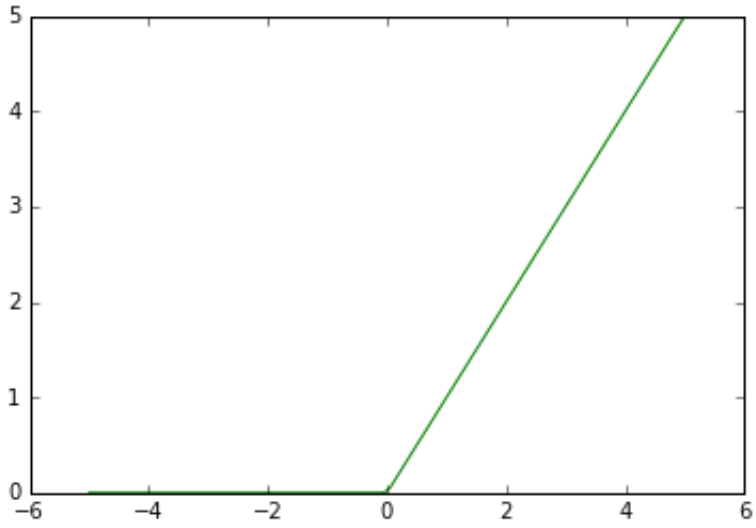
$$h_i = \mathrm{ReLU}(x_i) = \max(x_i, 0)$$

$$\mathbf{h} = \mathrm{ReLU}(\mathbf{x})$$

- Easier to optimize

In [5]:
```
xx = np.linspace(-5, 5, 100)
plt.plot(xx, xx * (xx > 0).astype(np.int), '-g')
```

Out[5]:
```
[<matplotlib.lines.Line2D at 0x7f25de355e50>]
```

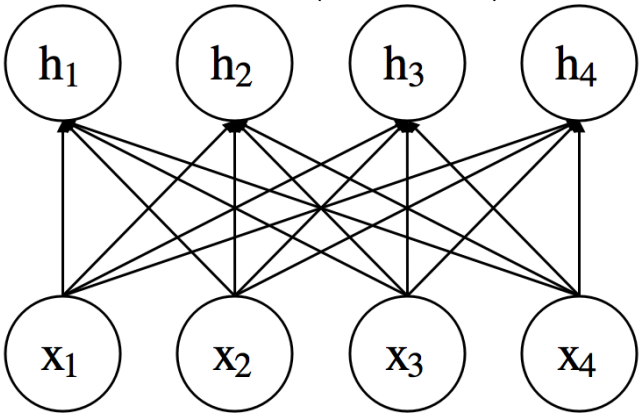


Types of Layers: Softmax

$$h_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

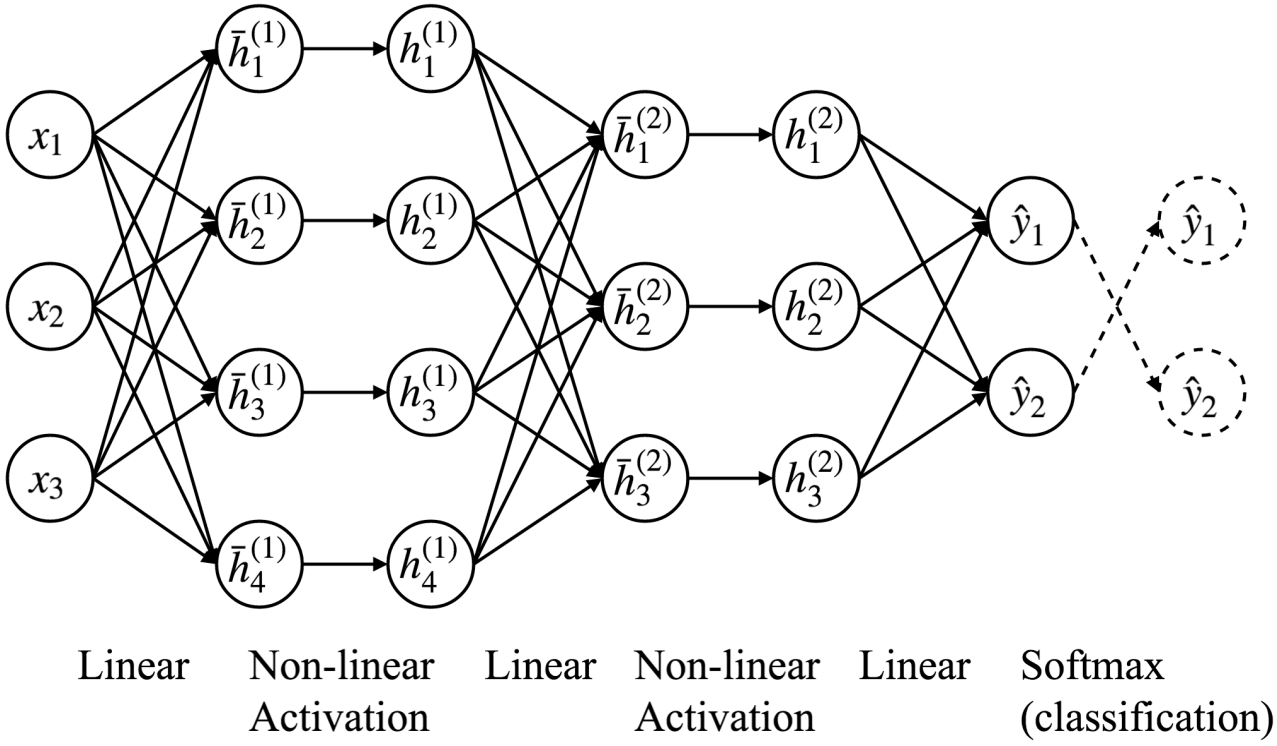
$$\mathbf{h} = \mathrm{Softmax}(\mathbf{x})$$

- Note: $h_i \geq 0$ and $\sum_i h_i = 1$
- Useful for generating a multinomial distribution (classification)



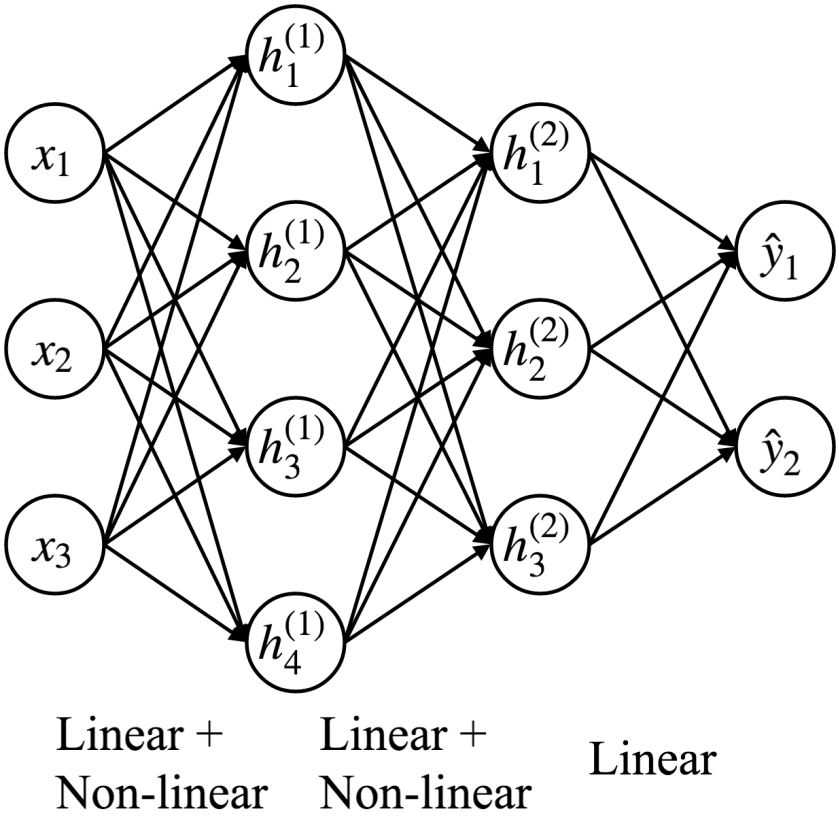
Multi-layer Neural Network

- Consists of multiple (linear + non-linear activation) layers.
- Each layer learns non-linear features from its previous layer.
- Often called Multi-Layer Perceptron (MLP).
- 2-layer MLP with infinite number of hidden units can approximate any functions.



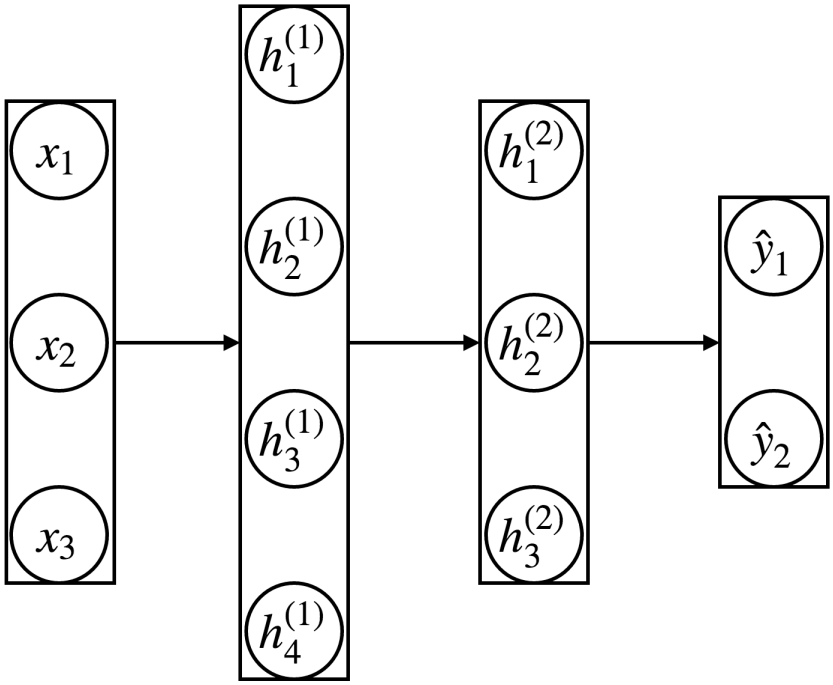
Multi-layer Neural Network

- Simplified illustration that only shows edges with weights.
- We assume that each layer is followed by a non-linear activation function (except for the output layer).



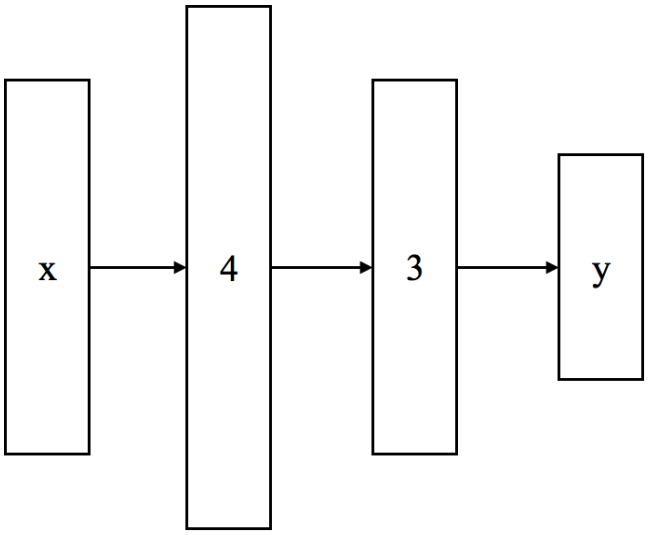
Multi-layer Neural Network

- More simplified illustration



Multi-layer Neural Network

- Even more simplified illustration



Outline

- Motivation
- Basics of Neural Networks
  - Forward Propagation
  - **Backward Propagation**
- Deep Neural Networks
  - Convolutional Neural Networks
  - Recurrent Neural Networks
- Applications
  - Computer Vision
  - Natural Language Processing
  - Reinforcement Learning

Training Neural Networks

- Repeat until convergence
  - $(\mathbf{x}, \mathbf{y})$ ← Sample an example (or a mini-batch) from data
  - $\hat{\mathbf{y}} \leftarrow f(\mathbf{x}; \theta)$ Forward propagation
  - Compute $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$
  - $\nabla_\theta \mathcal{L}$ ← Backward propagation
  - Update weights using (stochastic) gradient descent
    - $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$

Types of Losses: Squared Loss

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$$

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \hat{y}_i - y_i \iff \nabla_{\hat{\mathbf{y}}} \mathcal{L} = \hat{\mathbf{y}} - \mathbf{y}$$

- Used for regression problems

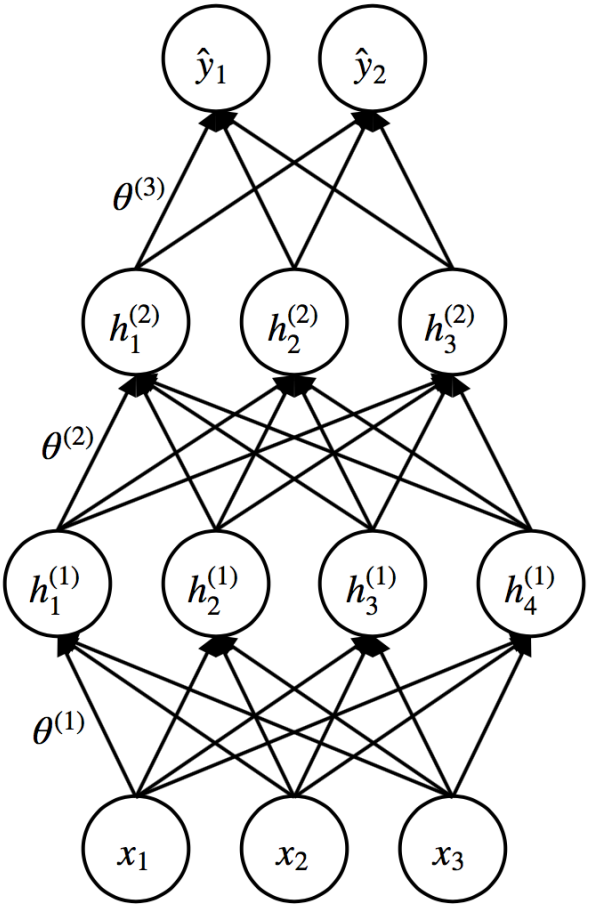
Types of Losses: Cross Entropy

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log \hat{y}_i$$

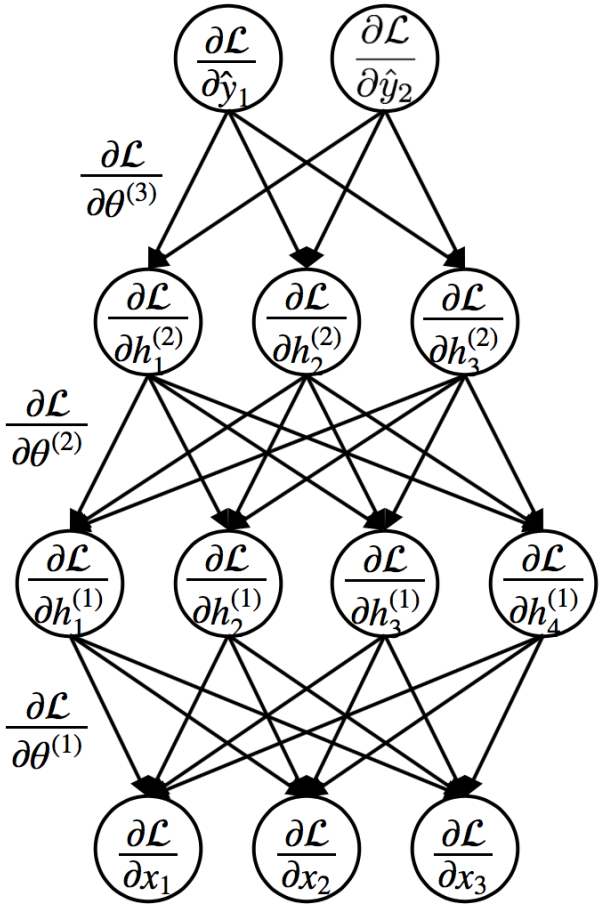
$$\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} \iff \nabla_{\hat{\mathbf{y}}} \mathcal{L} = -\frac{\mathbf{y}}{\hat{\mathbf{y}}}$$

- Measure a distance between two multinomial distributions
- Used for classification problems
  - ex) $\mathbf{y}$ =[0 0 1], $\hat{\mathbf{y}}$ =[0.3 0.2 0.5], $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\log 0.5$
- When cross entropy is used with a softmax output, the last layer is equivalent to softmax regression (multi-class version of logistic regression).

Training Neural Networks



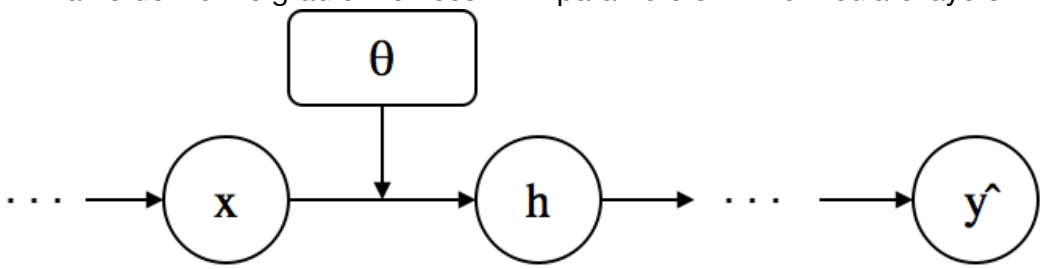
Forward



Backward

Idea of Back-Propagation

- Denote x, h, θ is the input, output, and parameter of a layer.
- It is non-trivial to derive the gradient of loss w.r.t. parameters in intermediate layers

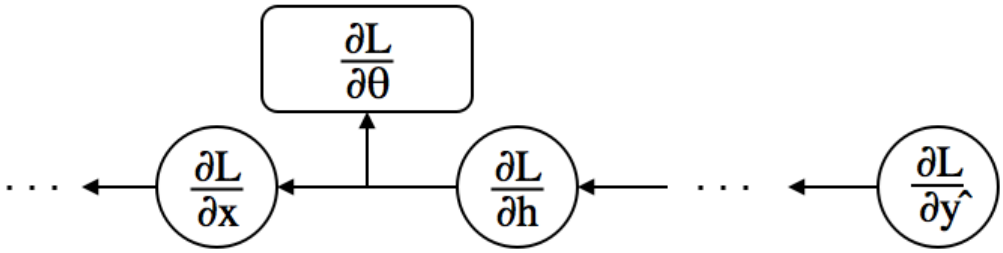


Idea of Back-Propagation

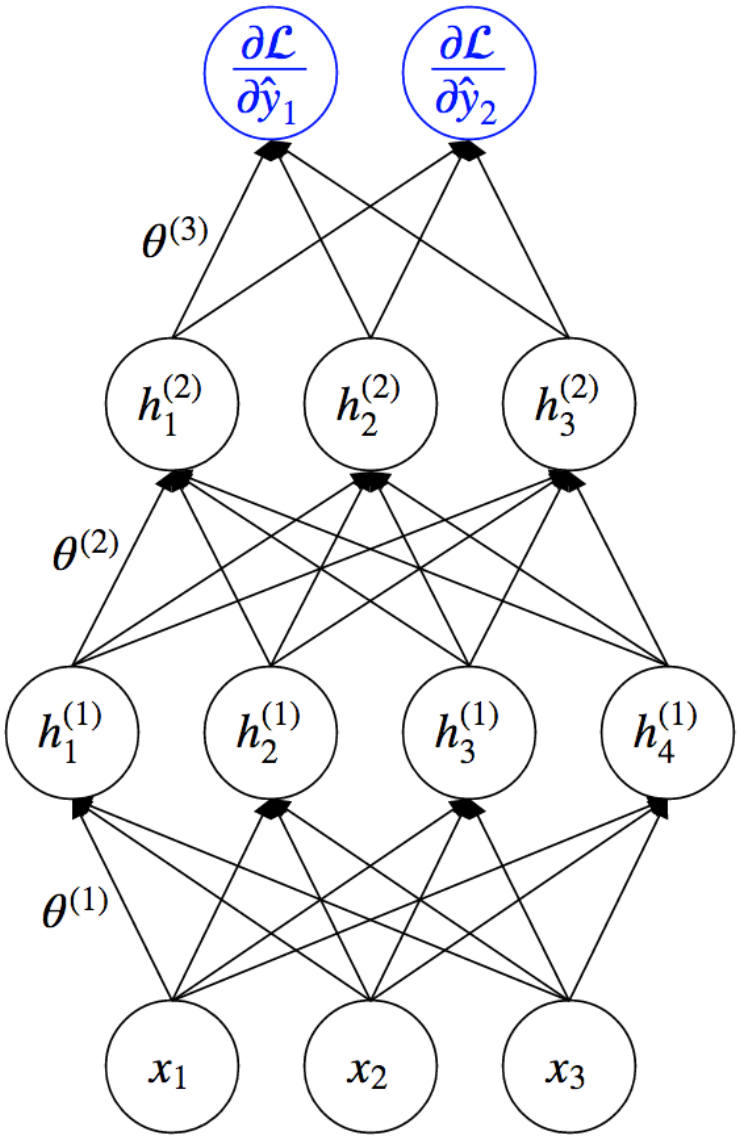
- Assuming that $\frac{\partial \mathcal{L}}{\partial h}$ is given, use the **chain rule** to compute the gradients

$$\frac{\partial \mathcal{L}}{\partial \theta} = \underbrace{\frac{\partial \mathcal{L}}{\partial h}}_{\text{given}} \underbrace{\frac{\partial h}{\partial \theta}}_{\text{easy}}, \frac{\partial \mathcal{L}}{\partial x} = \underbrace{\frac{\partial \mathcal{L}}{\partial h}}_{\text{given}} \underbrace{\frac{\partial h}{\partial x}}_{\text{easy}}$$

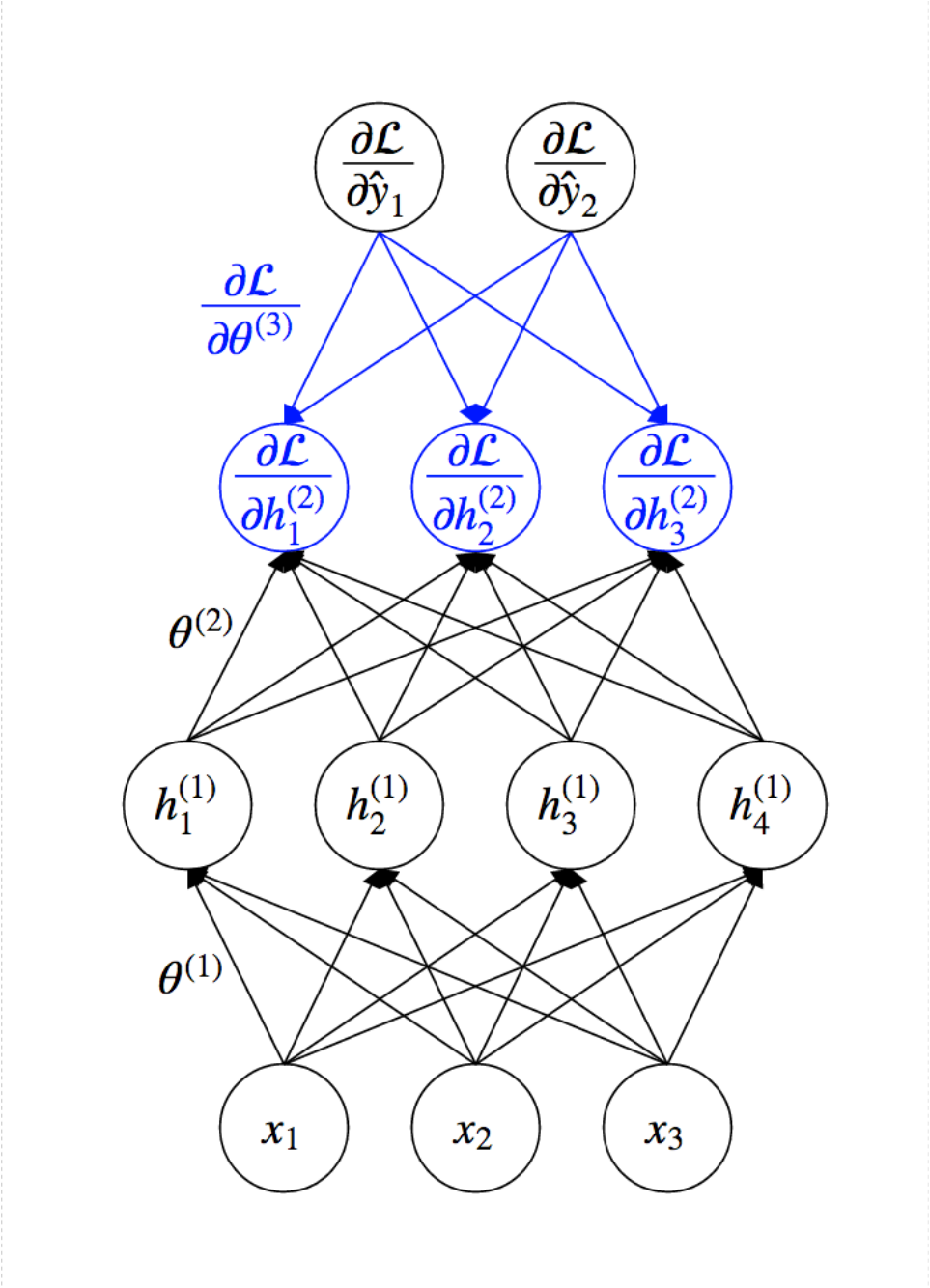
- We need only $\frac{\partial \mathcal{L}}{\partial \theta}$ for gradient descent. Why compute $\frac{\partial \mathcal{L}}{\partial x}$?
  - The previous layer needs it because x is the output of the previous layer.



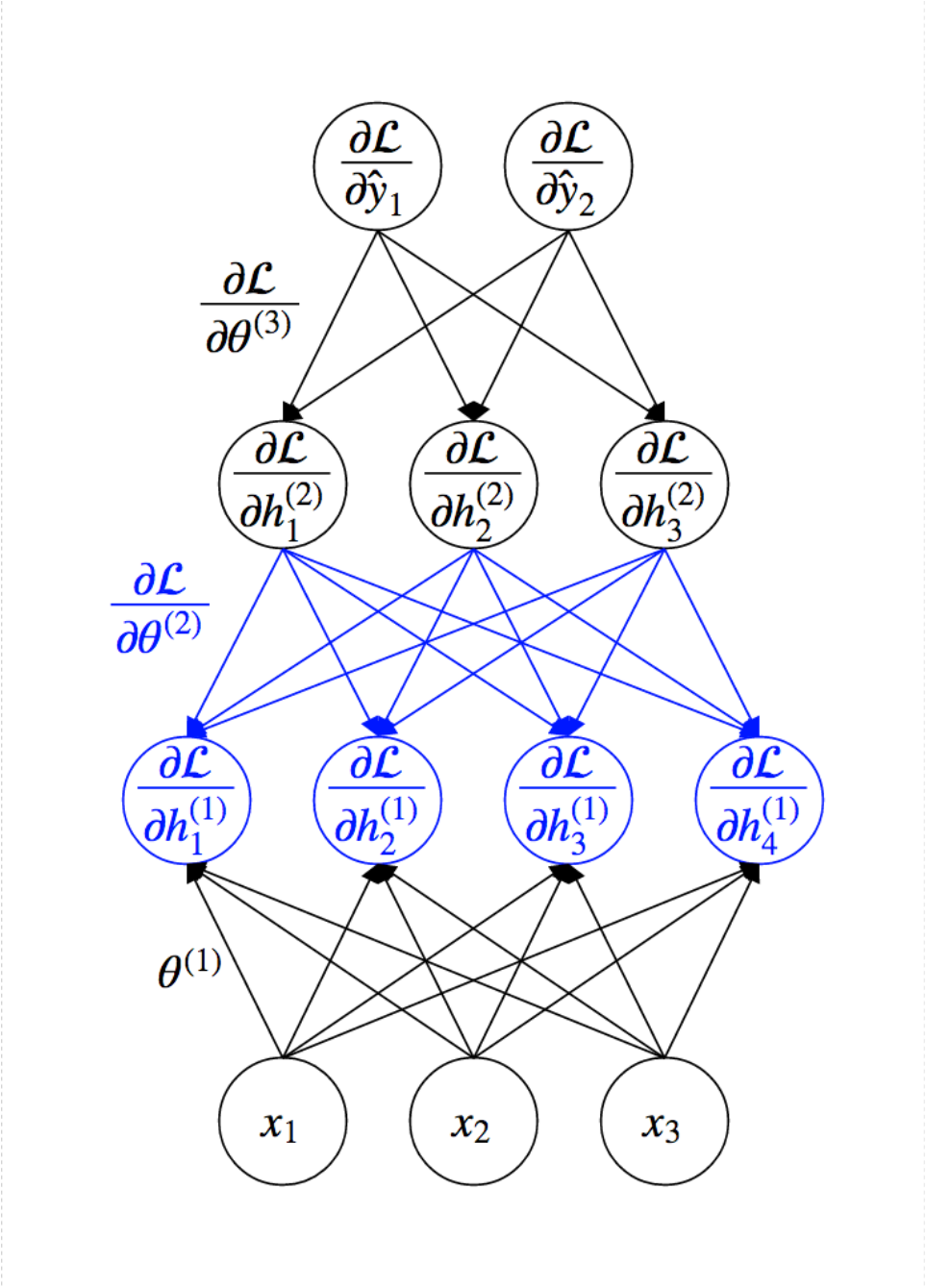
Idea of Back-Propagation



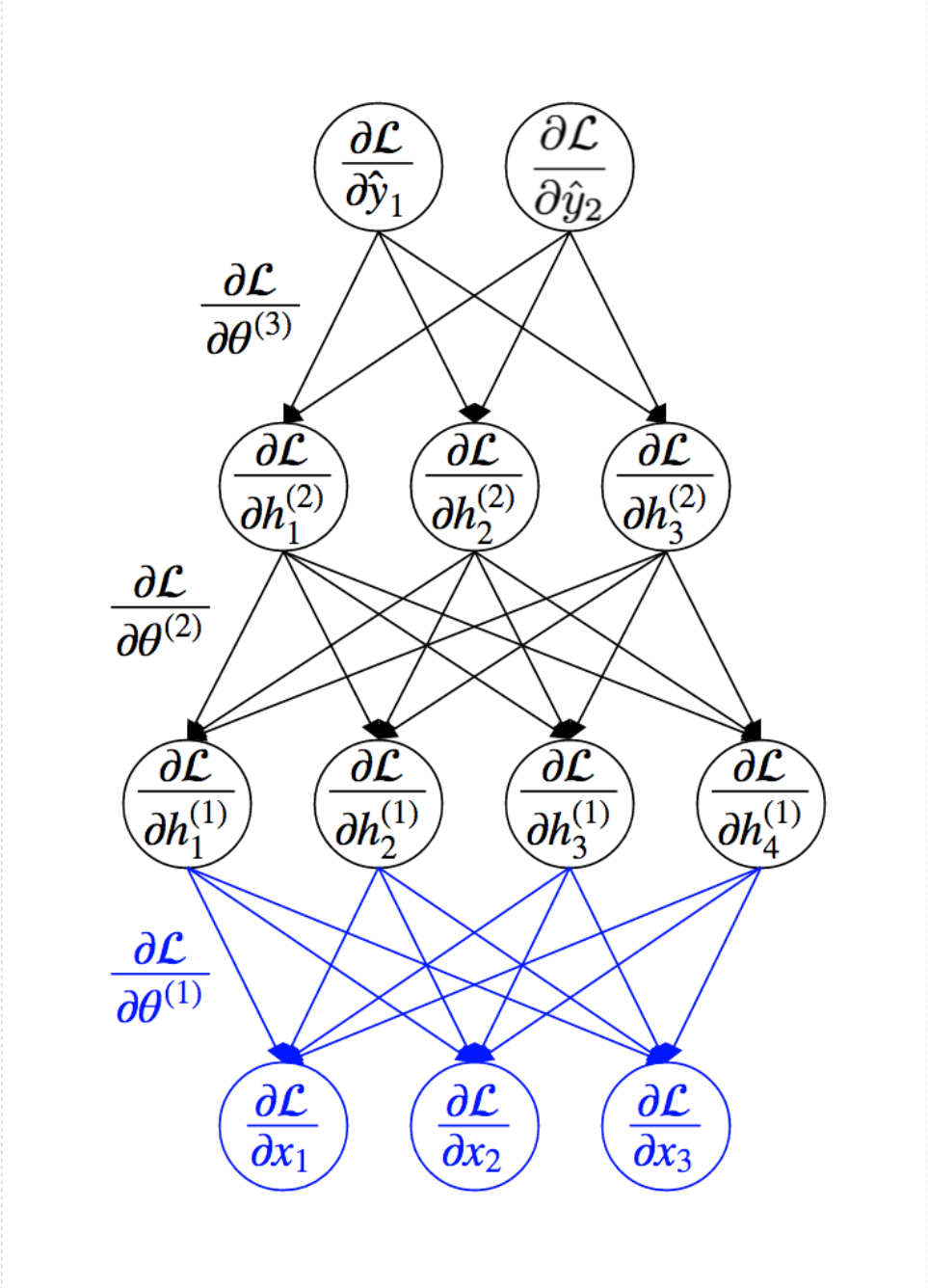
Idea of Back-Propagation



Idea of Back-Propagation



Idea of Back-Propagation



Back-Propagation Algorithm

- Compute $\nabla_{\mathbf{y}} \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial y_1}, \ldots, \frac{\partial \mathcal{L}}{\partial y_n} \right]$ directly from the loss function.
- For each layer (from top to bottom) with output $\mathbf{h}$, input $\mathbf{x}$, and weights $\mathbf{W}$,
  - Assuming that $\nabla_{\mathbf{h}} \mathcal{L}$ is given, compute gradients using the **chain rule** as follows:

$$\nabla_{\mathbf{W}} \mathcal{L} = \nabla_{\mathbf{h}} \mathcal{L} \nabla_{\mathbf{W}} \mathbf{h}$$

$$\nabla_{\mathbf{x}} \mathcal{L} = \nabla_{\mathbf{h}} \mathcal{L} \nabla_{\mathbf{x}} \mathbf{h}$$

Practice: Linear

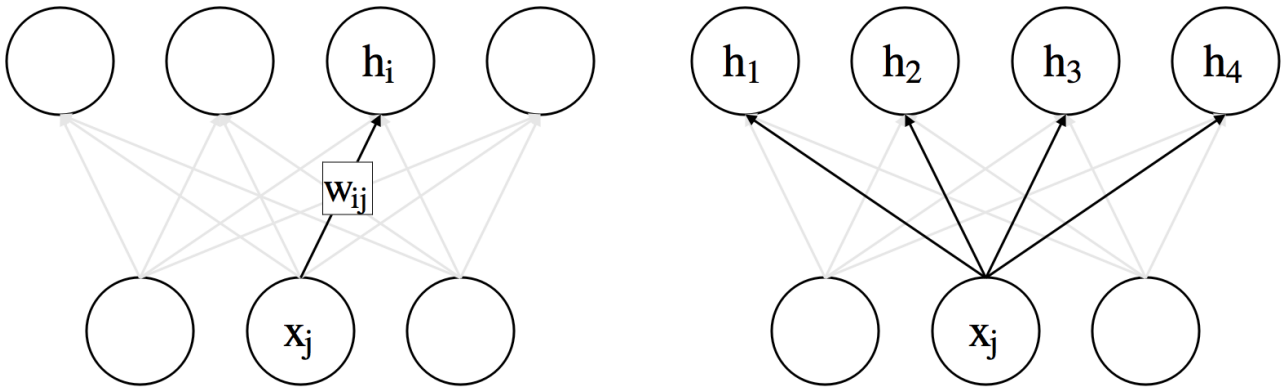
- Forward: $h_i = \sum_j w_{ij} x_j + b_i \iff \mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b}$
- Gradient w.r.t. parameters

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial h_i} \frac{\partial h_i}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial h_i} x_j \iff \nabla_{\mathbf{W}} \mathcal{L} = \nabla_{\mathbf{h}} \mathcal{L} \mathbf{x}^\top$$

$$\nabla_{\mathbf{b}} \mathcal{L} = \nabla_{\mathbf{h}} \mathcal{L}$$

- Gradient w.r.t. inputs

$$\frac{\partial \mathcal{L}}{\partial x_j} = \sum_i \frac{\partial \mathcal{L}}{\partial h_i} \frac{\partial h_i}{\partial x_j} = \sum_i \frac{\partial \mathcal{L}}{\partial h_i} w_{ij} \iff \nabla_{\mathbf{x}} \mathcal{L} = \mathbf{W}^\top \nabla_{\mathbf{h}} \mathcal{L}$$

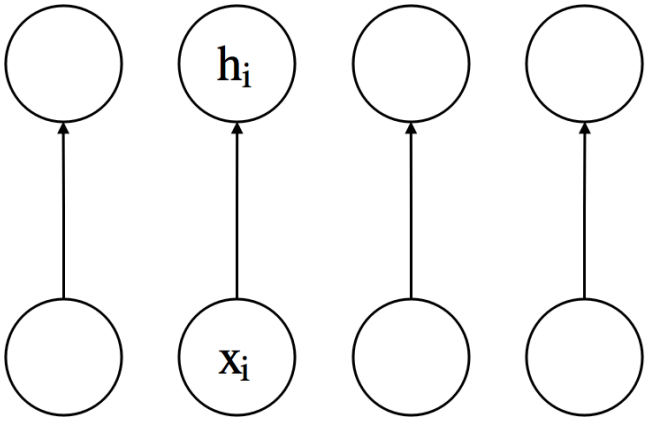


Practice: Non-Linear Activation (Sigmoid)

- Forward: $h_i = \sigma(x_i) = \frac{1}{1+\exp(-x_i)} \iff \mathbf{h} = \sigma(\mathbf{x})$
- Gradient w.r.t. inputs

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial h_i}\frac{\partial h_i}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial h_i}\sigma(x_i)(1-\sigma(x_i)) = \frac{\partial \mathcal{L}}{\partial h_i}h_i(1-h_i)$$

$$\nabla_{\mathbf{x}}\mathcal{L} = \nabla_{\mathbf{h}}\mathcal{L} \odot \mathbf{h} \odot (\mathbf{1}-\mathbf{h})$$



Example Code (Torch/Lua)

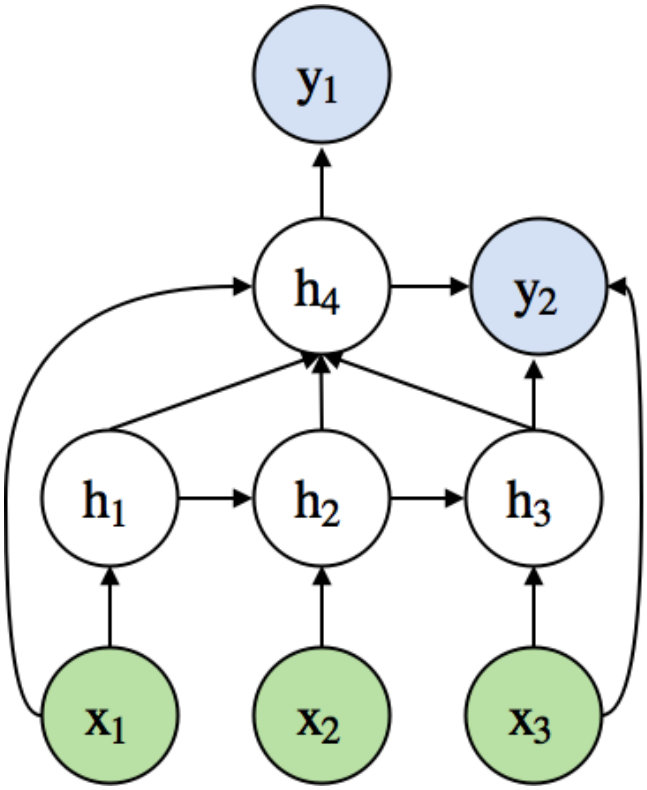
```
require "torch"
require "nn"

dataX = torch.Tensor(1000, 3)
dataY = torch.Tensor(1000):random(2)
-- network construction
-- 3 -> 4 -> 3 -> 2
model = nn.Sequential()
model:add(nn.Linear(3,4))
model:add(nn.Sigmoid())
model:add(nn.Linear(4,3))
model:add(nn.Sigmoid())
model:add(nn.Linear(3,2))
model:add(nn.LogSoftMax())
-- loss function (cross entropy)
criterion = nn.ClassNLLCriterion()
```

```
for iter=1,10000 do
    -- sample an example from a dataset
    idx = torch.random(1000)
    x = dataX[idx]
    y = dataY[idx]
    -- forward propagation
    y_pred = model:forward(x)
    loss = criterion:forward(y_pred, y)
    -- backward propagation
    model:zeroGradParameters() -- dL/dW = 0
    -- compute dL/dy
    dL_dy = criterion:backward(y_pred, y)
    -- compute gradient w.r.t. weights
    model:backward(x, dL_dy)
    -- SGD update with learning rate of 0.001
    model:updateParameters(0.001)
end
```

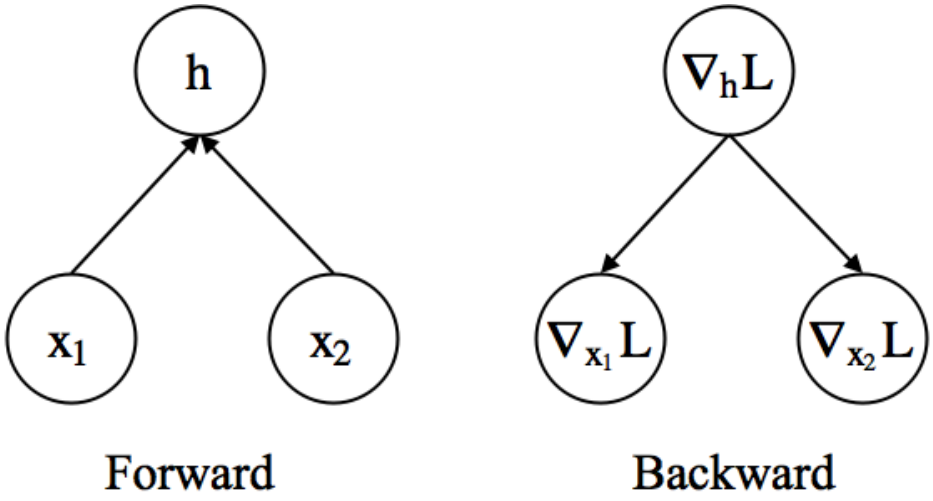
FAQ: Does a neural network have to be always layer-structured?

- No. It can be any directed acyclic graph (DAG).
- Example of a complex neural network



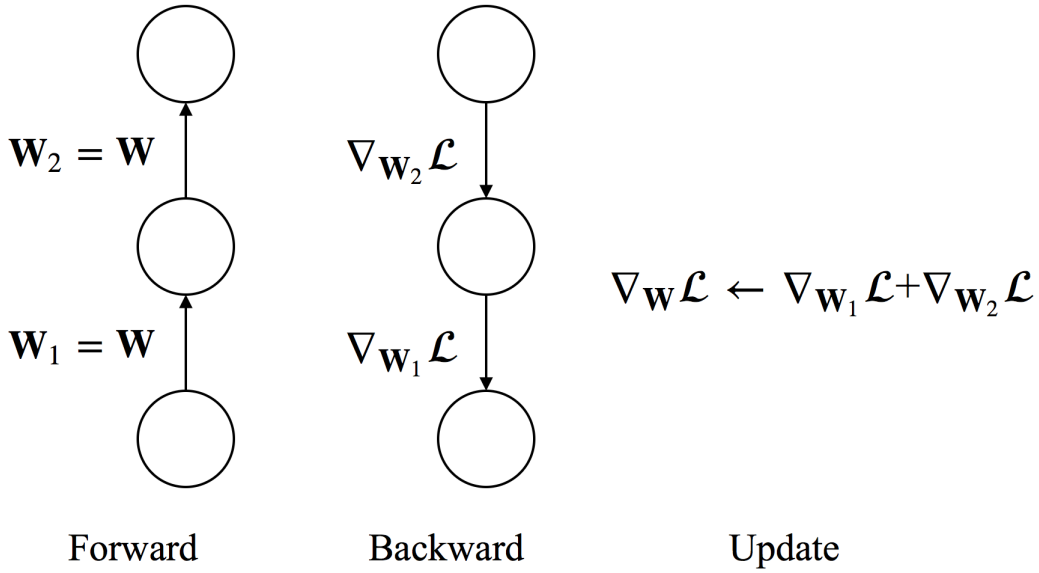
FAQ: Can we define any arbitrary layers?

- We can define any layer as long as it is differentiable.
- Example) Addition layer
  - Forward: $\mathbf{h} = \mathbf{x_1} + \mathbf{x_2}$
  - Backward
    - $\nabla_{\mathbf{x_1}} \mathcal{L} = \nabla_{\mathbf{h}} \mathcal{L} \nabla_{\mathbf{x_1}} \mathbf{h} = \nabla_{\mathbf{h}} \mathcal{L}$
    - $\nabla_{\mathbf{x_2}} \mathcal{L} = \nabla_{\mathbf{h}} \mathcal{L} \nabla_{\mathbf{x_2}} \mathbf{h} = \nabla_{\mathbf{h}} \mathcal{L}$



FAQ: How to handle shared weights?

- To constrain $W_1 = W_2 = W$, we need $\Delta W_1 = \Delta W_2$.
- Compute $\nabla_{W_1} \mathcal{L}$ and $\nabla_{W_2} \mathcal{L}$ separately.
- Use $\nabla_W \mathcal{L} = \nabla_{W_1} \mathcal{L} + \nabla_{W_2} \mathcal{L}$ to update the shared weight.
- In practice, we accumulate gradients to the shared memory space for $\nabla_W \mathcal{L}$ during back-propagation.
- Weight sharing is used in *convolutional neural networks* and *recurrent neural networks*.

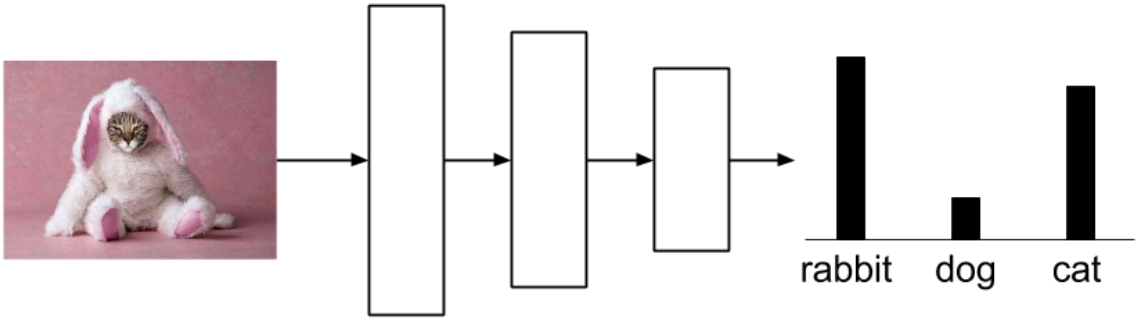


Outline

- Motivation
- Basics of Neural Networks
  - Forward Propagation
  - Backward Propagation
- **Deep Neural Networks**
  - Convolutional Neural Networks
  - Recurrent Neural Networks
- Applications
  - Computer Vision
  - Natural Language Processing
  - Reinforcement Learning

What is "deep" neural network?

- A neural network is considered to be **deep** if it has more than two (non-linear) hidden layers..
- Higher layers extract more abstract and hierarchical features.
  - In a classification network, features become more *linearly separable* as layer goes up



What is "deep" neural network?

- Difficulties in training deep neural networks
  - Easy to overfit (The number of parameters is large)
  - Hard to optimize (highly non-convex optimization)
  - Computationally expensive (many matrix multiplications)
- Recent Advances
  - Large-scale dataset (e.g., 1M images in ImageNet)
  - Better regularization (e.g., Dropout)
  - Better optimization (e.g., RMSProp, ReLU)
  - Better hardware (GPU for matrix computation)

Popular Deep Architectures

- Convolutional Neural Network (CNN)
  - Widely used for image modeling
  - ex) object recognition, segmentation, vision-based reinforcement learning problems
- Recurrent Neural Network (RNN)
  - Widely used for sequential data modeling
  - ex) machine translation, image caption generation

Outline

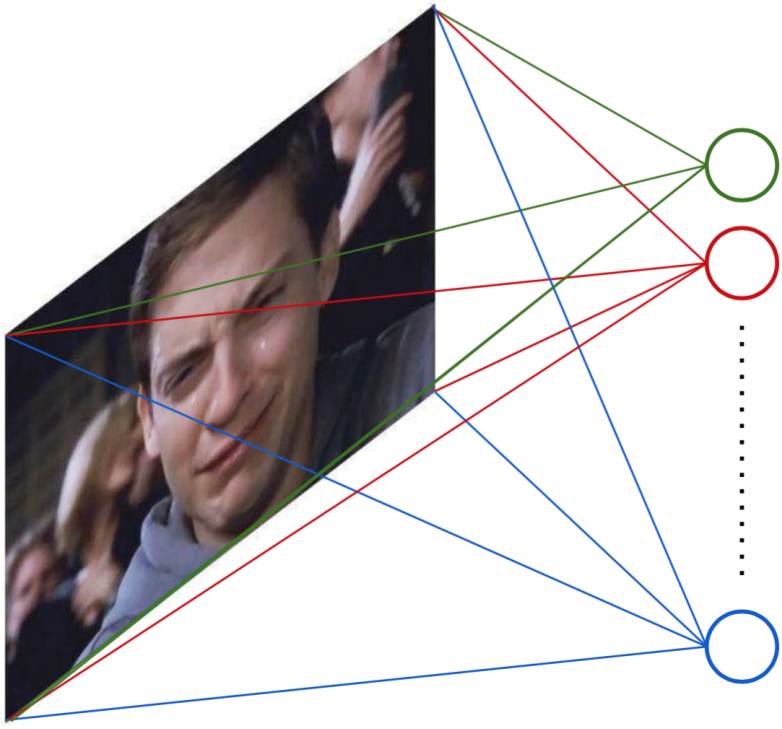
- Motivation
- Basics of Neural Networks
  - Forward Propagation
  - Backward Propagation
- Deep Neural Networks
  - **Convolutional Neural Networks**
  - Recurrent Neural Networks
- Applications
  - Computer Vision
  - Natural Language Processing
  - Reinforcement Learning

Convolutional Neural Network

- A special kind of multi-layer neural network
- Designed to recognize visual patterns directly from raw pixels

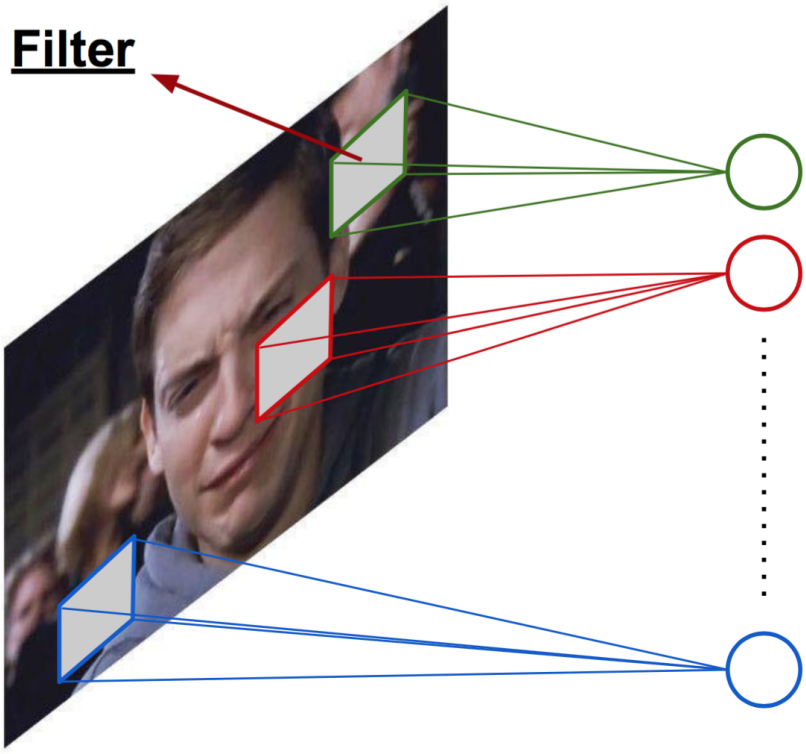
Multi-layer Perceptron (MLP)

- Consider 100x100 input pixels
- 1 hidden layer with 10000 hidden units
- **100M parameters** → infeasible! → Pixels are locally correlated!



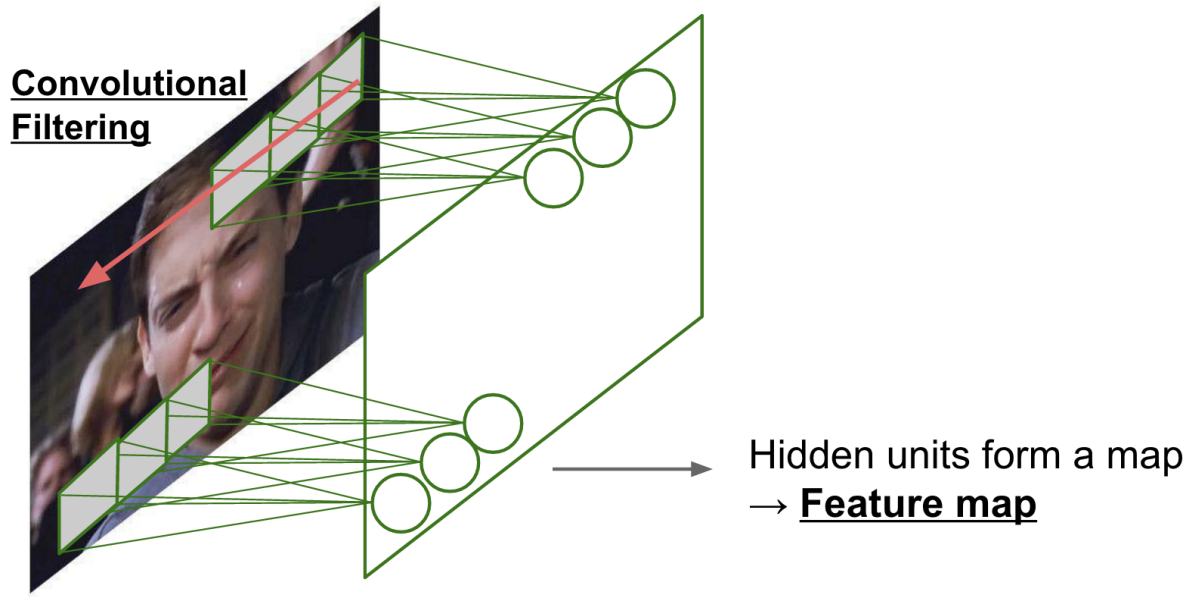
Locally-connected Neural Network

- Consider 100x100 input pixels
- Each unit is connected to 10x10 pixels.
- Each unit extracts a local pattern from the image.
- 10000 hidden units
- **1M parameters** → still too large



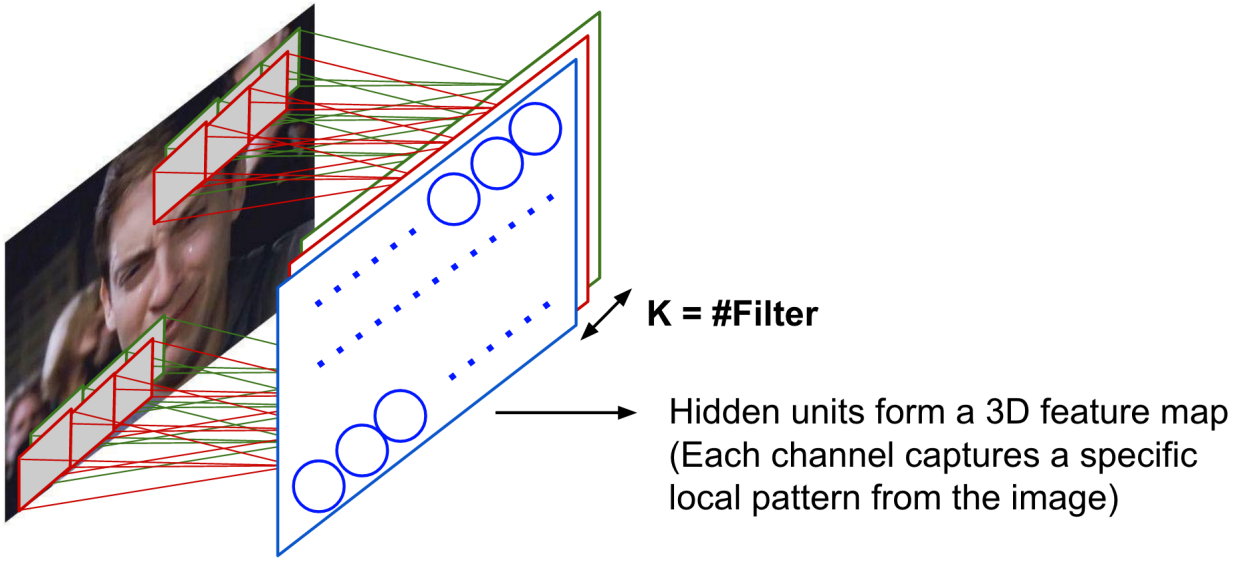
Convolutional Neural Network (one filter)

- Consider 100x100 input pixels
- Apply the **same filter (weight)** over the entire image.
- Hidden units form a 100x100 **feature map**.
- 10x10 parameters.
  → **only captures a single local pattern.**

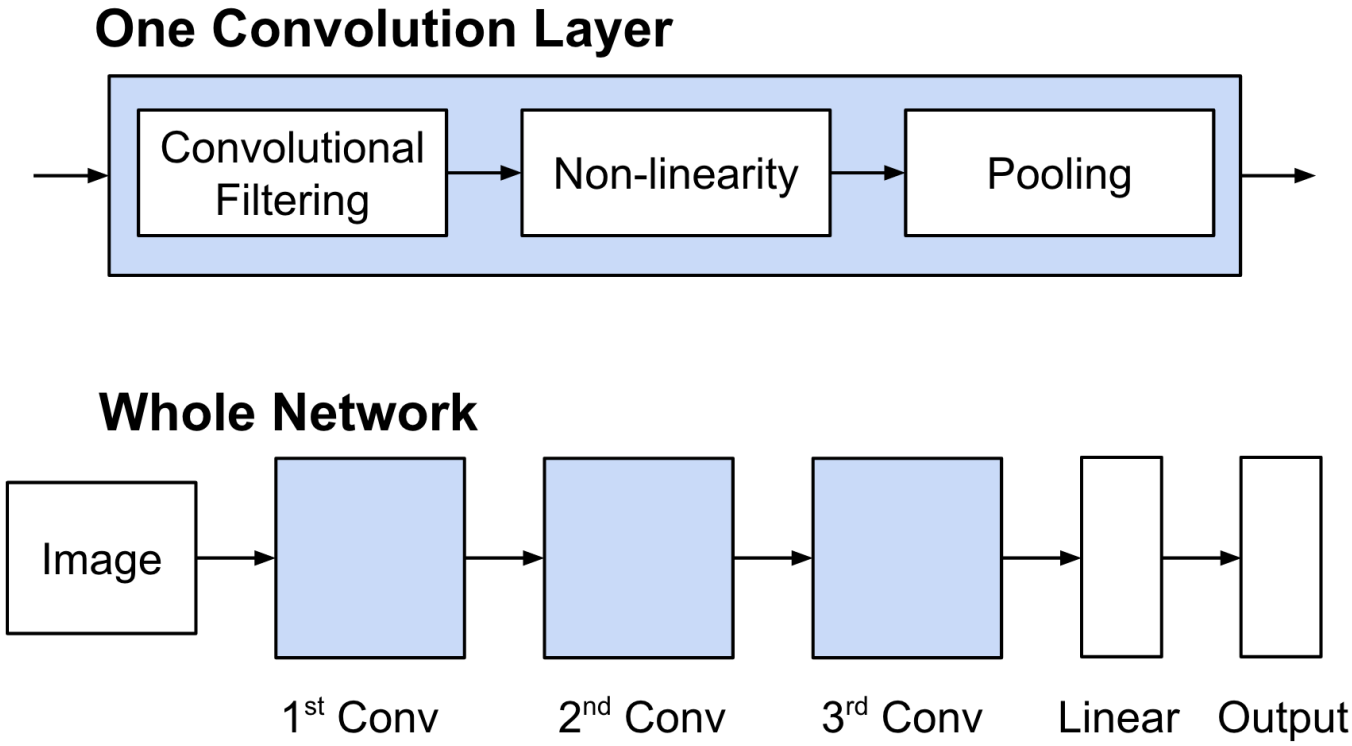


Convolutional Neural Network (multiple filters)

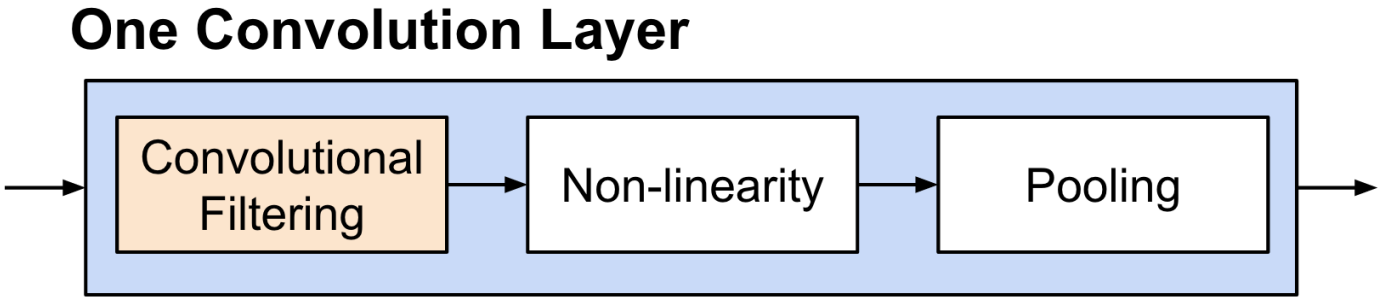
- 100x100 input pixels
- Apply K number of 10x10 filters.
- Hidden units form a Kx100x100 feature map.
- Kx10x10 parameters.
- Num of filters and size of filters are hyperparameters.



Typical Deep CNN Architecture

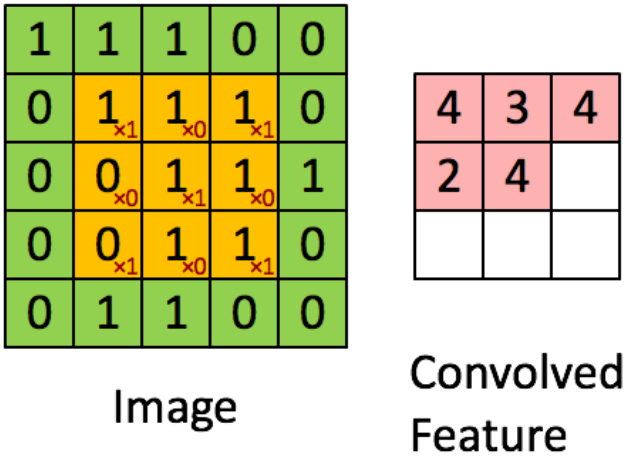


Details of Convolution



Details of Convolution: Convolutional Filtering

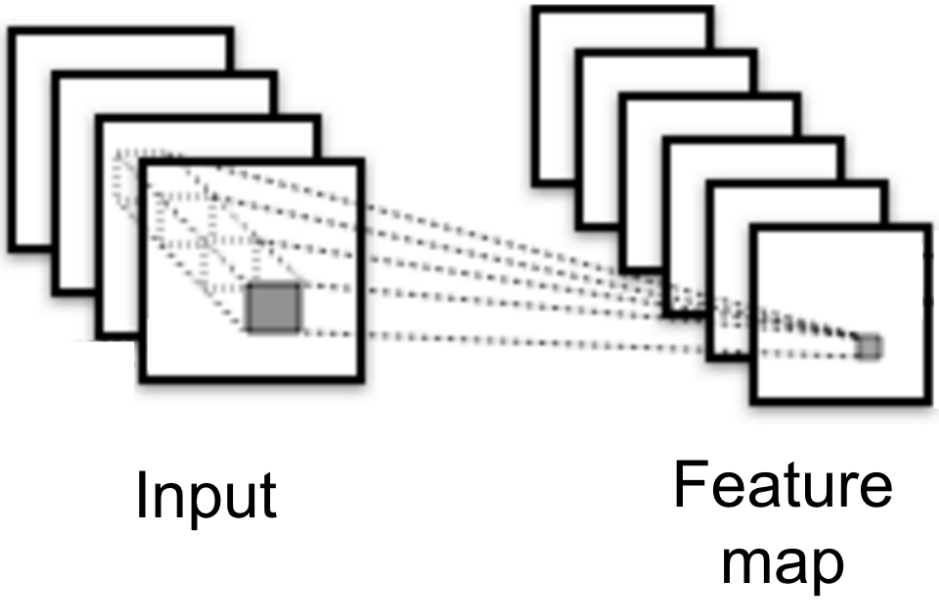
- A filter has $\mathbf{W} \in \mathbb{R}^{h \times w}$ weights (bias is omitted for simplicity).
- Compute inner products between $\mathbf{W}$ and $h \times w$ input patches by sliding window.
  - The same weight is shared across the entire image.
- The following animation shows the simplest case: one-channel input, one filter



(Figure from Stanford UFLDL Tutorial)

Details of Convolution: Convolutional Filtering

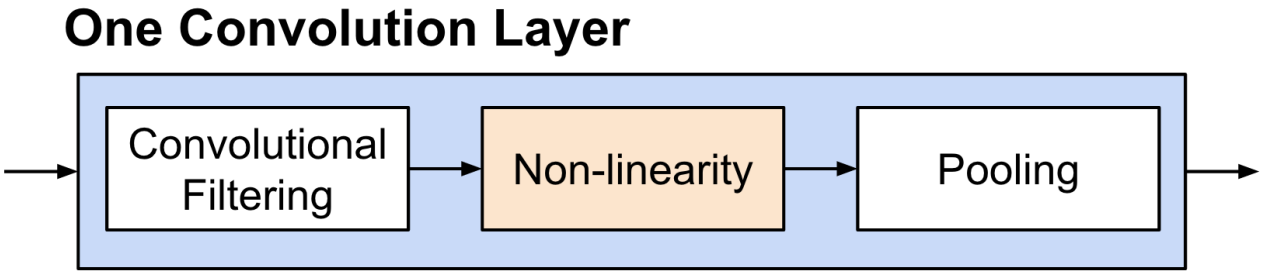
- In general, an input consists of multiple channels.
  - Each filter has $\mathbf{W} \in \mathbb{R}^{c \times h \times w}$ weight (c: #channels of input).
- Applying K different filters $\rightarrow$ produces a 3D feature map (stacked through channels).
- Hyperparameters: num of filters, size of filters
- The following figure shows the general case: multi-channel input, mutilple filters



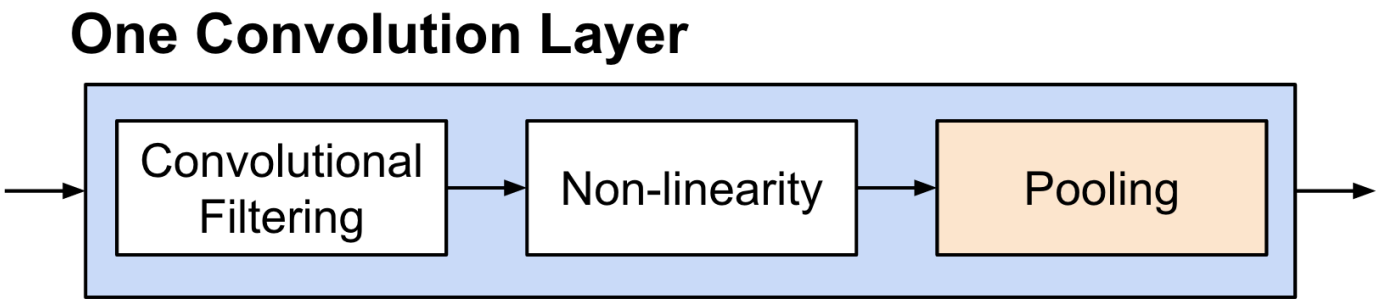
(Figure from Yann LeCun)

Details of Convolution: Non-linearity

- Method: Just apply non-linear function (e.g., Sigmoid, ReLU)
- ReLU is preferred because it is easier to optimize.

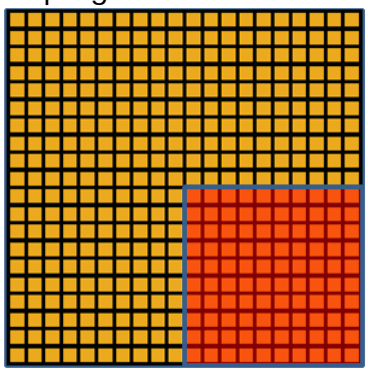


Details of Convolution: Pooling



Details of Convolution: Pooling

- Method: Take average or maximum over HxW region of input feature.
- Outcome
  - Shrink the number of hidden units.→ reduces the number of parameters at the end.
  - Make features robust to small translations of image.
- Hyperparameters: pooling method (avg or max), pooling size
- Often called "sub-sampling"



1	7
5	9

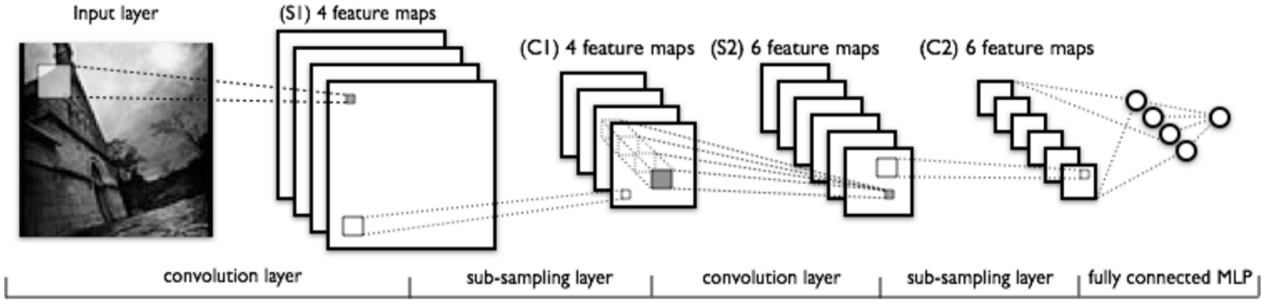
Convolved feature

Pooled feature

(Figure from Stanford UFLDL Tutorial)

Illustration of Deep CNN

- Feature maps become smaller in higher layers due to pooling.
- hidden units in higher layers capture patterns from larger input patches.



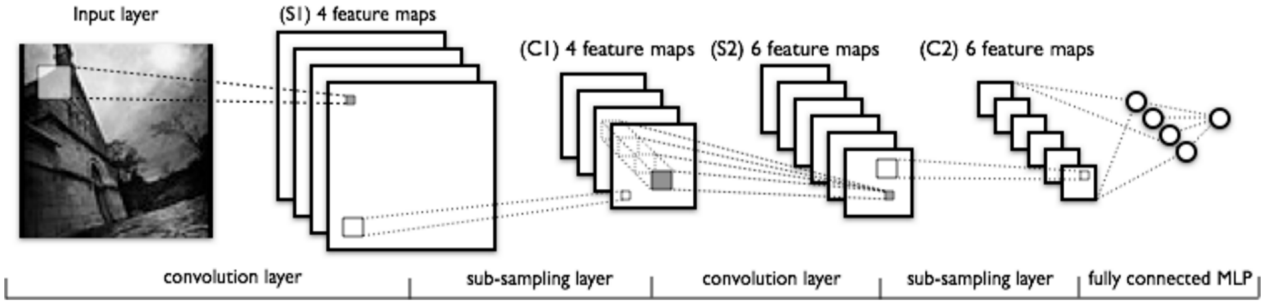
(Figure from Yann LeCun)

Example Code (Torch/Lua)

```
require "torch"
require "nn"
model = nn.Sequential()
-- 1st convolution: 3 -> 64 (5x5 filters)
model:add(nn.SpatialConvolution(3, 64, 5, 5))
model:add(nn.ReLU())
model:add(nn.SpatialMaxPooling(2,2,2,2)) -- 2x2 pooling
-- 2nd convolution: 64 -> 128 (5x5 filters)
model:add(nn.SpatialConvolution(64, 128, 5, 5))
model:add(nn.ReLU())
model:add(nn.SpatialMaxPooling(2,2,2,2))
-- Reshape 3D map (128*5*5) to a long vector
model:add(nn.View(-1):setNumInputDims(3))
-- 3rd linear: 128*5*5 -> 128 hidden units
model:add(nn.Linear(128*5*5, 128))
model:add(nn.ReLU())
-- 4th linear: 128 -> 10 classes
model:add(nn.Linear(128, 10))
model:add(nn.LogSoftMax()))
```

What is learned by CNN? Filter Visualization

- Train a deep CNN on *ImageNet* (1.2M images, 1000 classes)
- Perform forward propagations from many examples
- Find image patches that strongly activate a specific feature map (filter)
- Reconstruct the input patch from the feature map
- Proposed by Zeiler and Fergus (ECCV 2014)

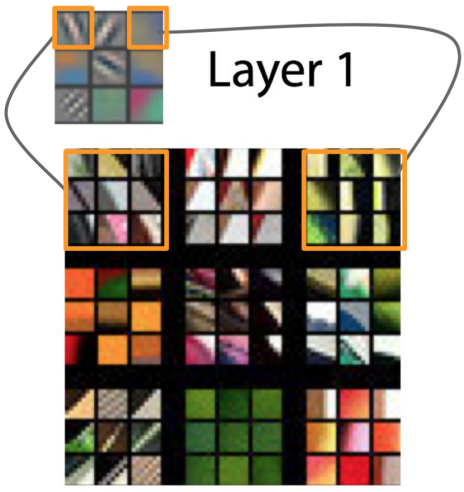


(Figure from Yann LeCun)

Filter Visualization: 1st and 2nd Layer

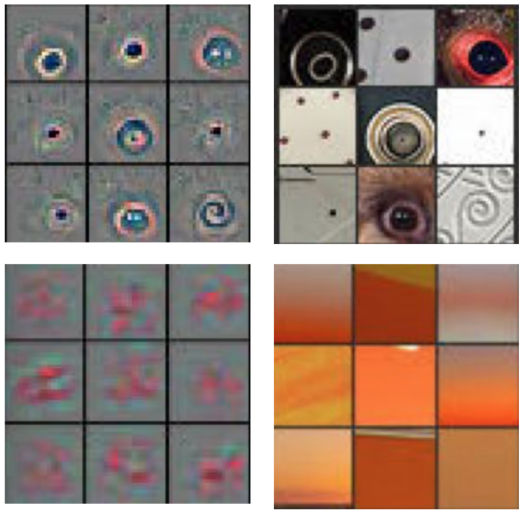
- Layer 1

Specific edges



- Layer 2

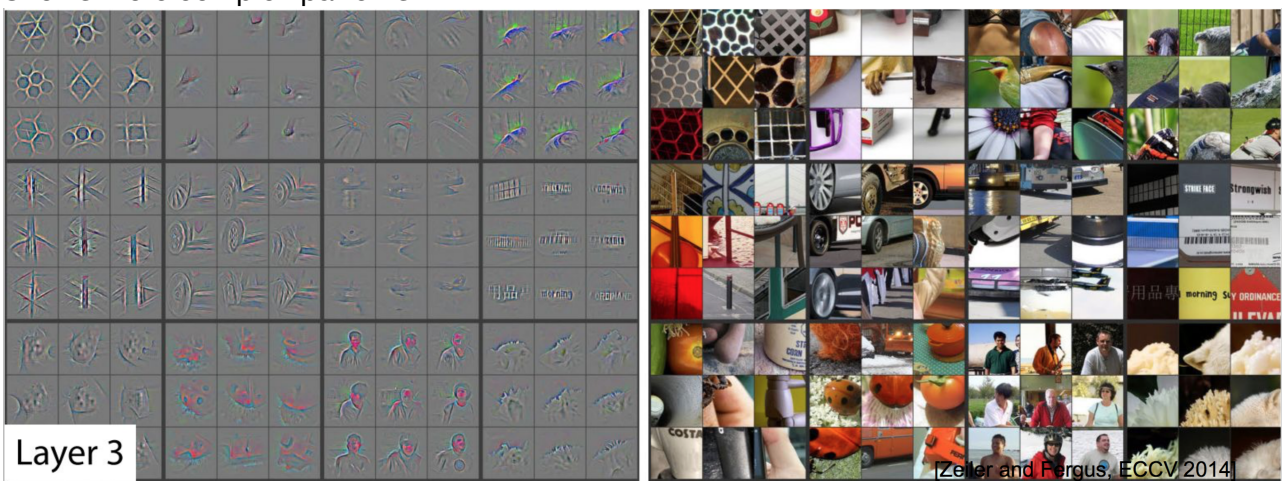
Conjunction of edge/color



[Zeiler and Fergus, ECCV 2014]

Filter Visualization: 3rd Layer

- Shows more complex patterns



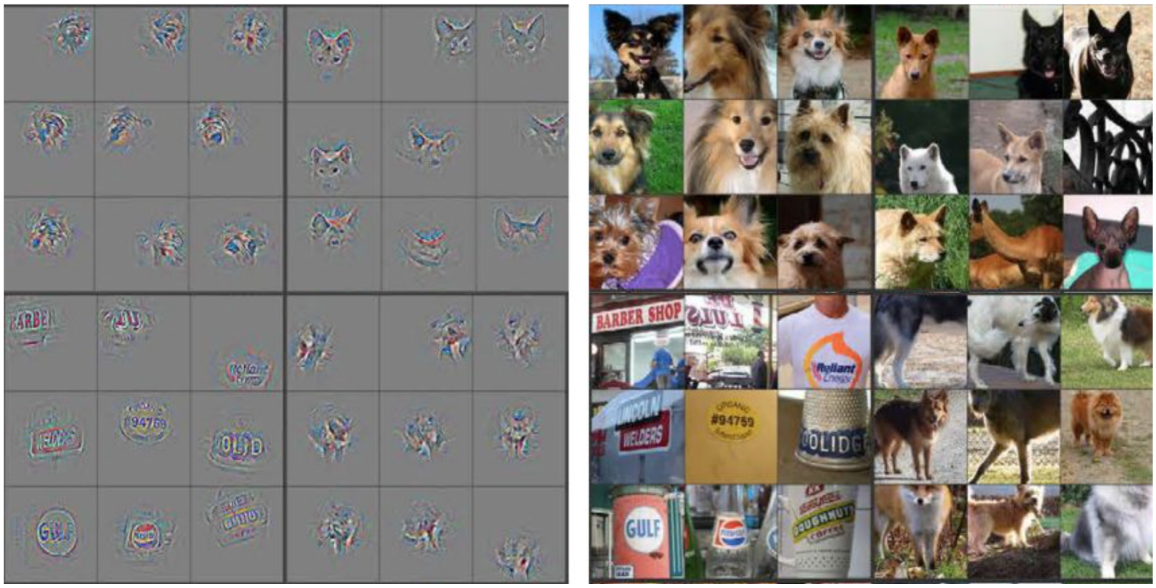
Filter Visualization: 4th Layer

- More class-specific

[Zeiler and Fergus, ECCV 2014]

Filter Visualization: 5th Layer

- Shows entire objects with pose variations
- Each filter can be viewed as a part detector. (e.g., dog face, text, animal leg)

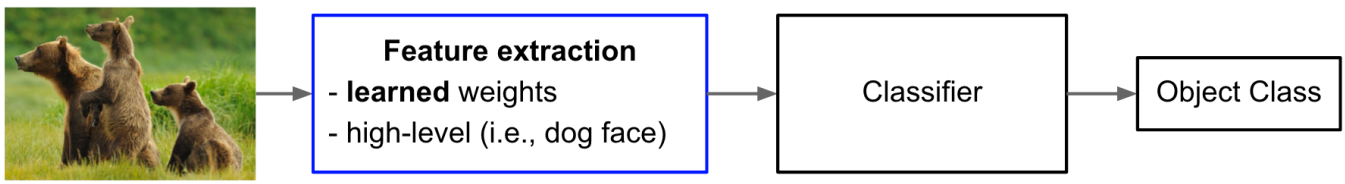


[Zeiler and Fergus, ECCV 2014]

Comparison to Traditional Approach

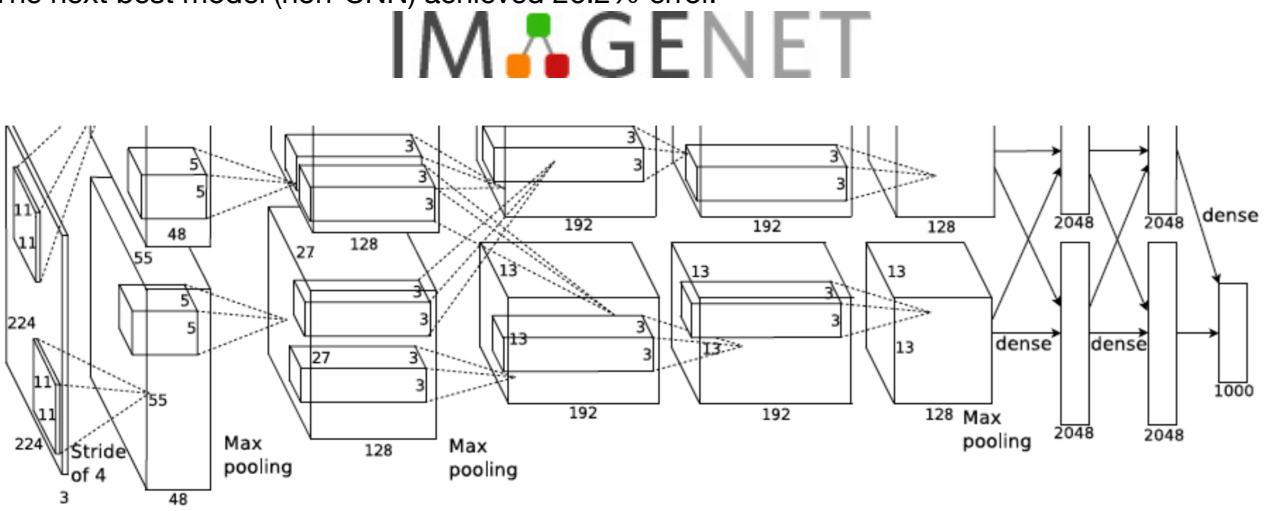
- Traditional Computer Vision Approach

- Convolutional Neural Networks



ImageNet Challenge 2012

- AlexNet (7 layers, Krizhevsky et al.) achieved 16.4% error.
- The next best model (non-CNN) achieved 26.2% error.



(Figure from Alex Krizhevsky et al.)

ImageNet Challenge

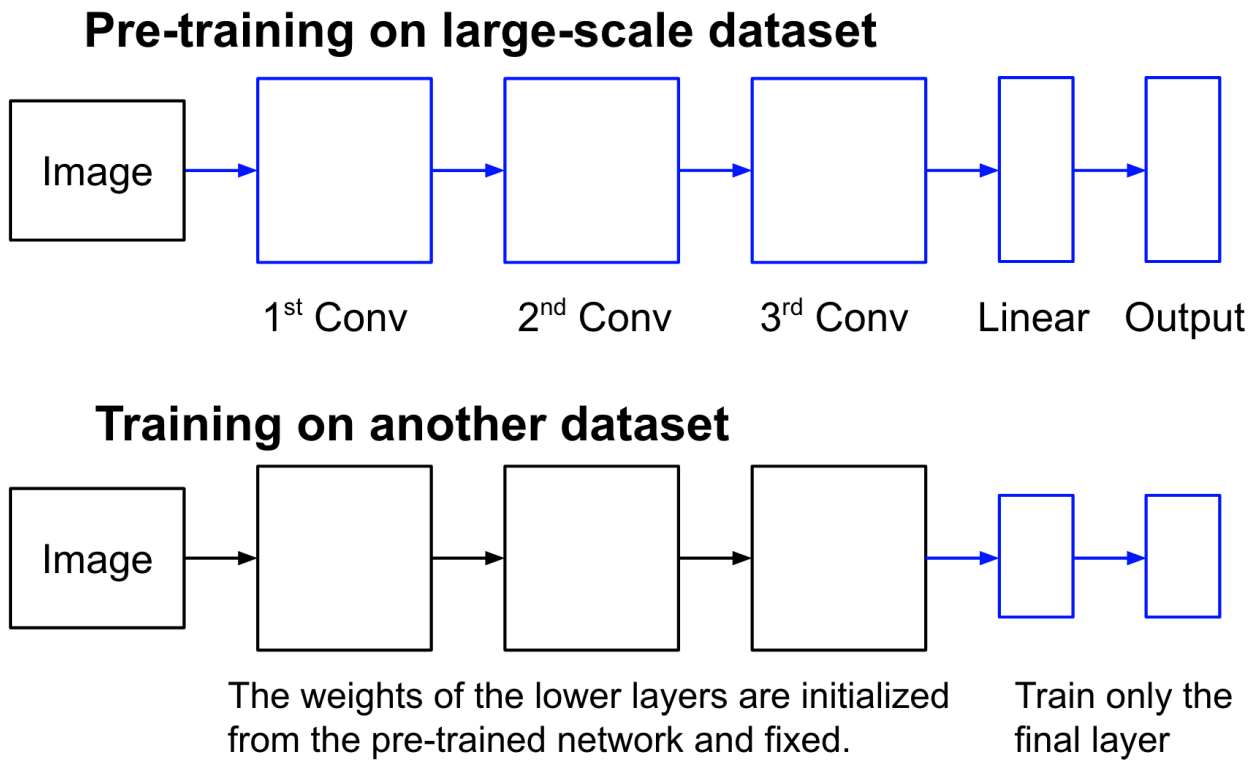
- ImageNet 2013: Clarifi (7 layers) → 14.8% error

- ImageNet 2014: GoogLeNet (22 layers) → 4.9% error (Human: 5.1% error)

- ImageNet 2015: ResNet (152 layers!) → 3.5% error

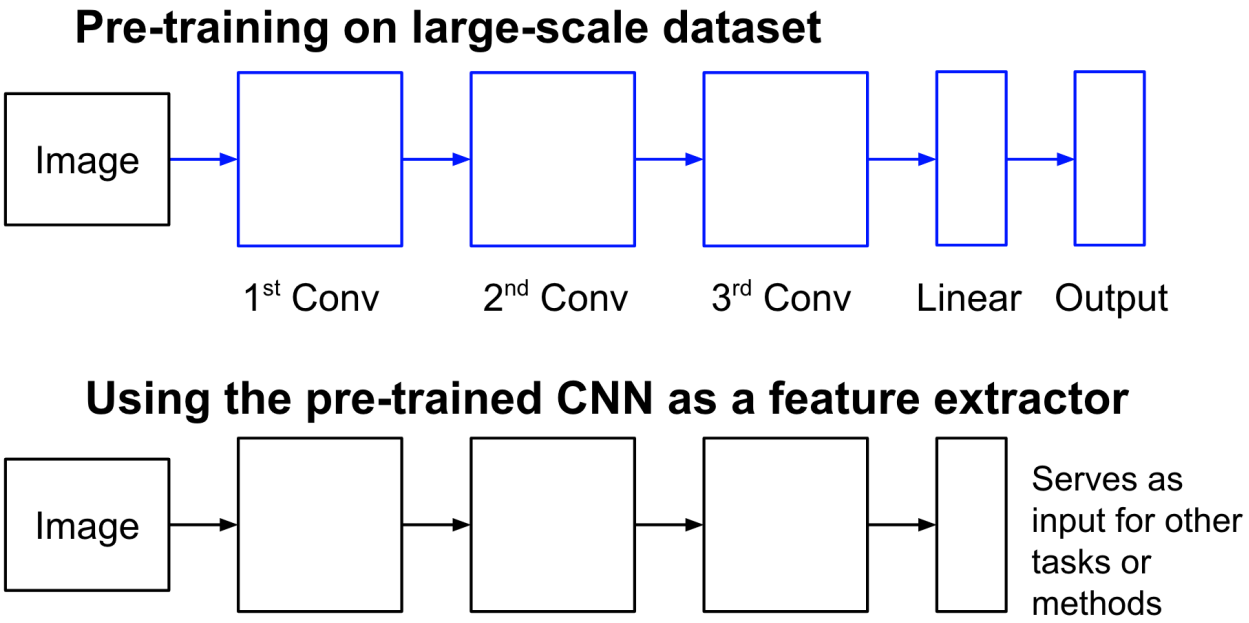
Feature Generalization: Dataset

- Pre-train a CNN on a large-scale dataset (ImageNet) and **train only the final linear layer** on another dataset.
- Achieves state-of-the-art results on small datasets (e.g., Caltech-101).
- The learned features can generalize to any dataset.



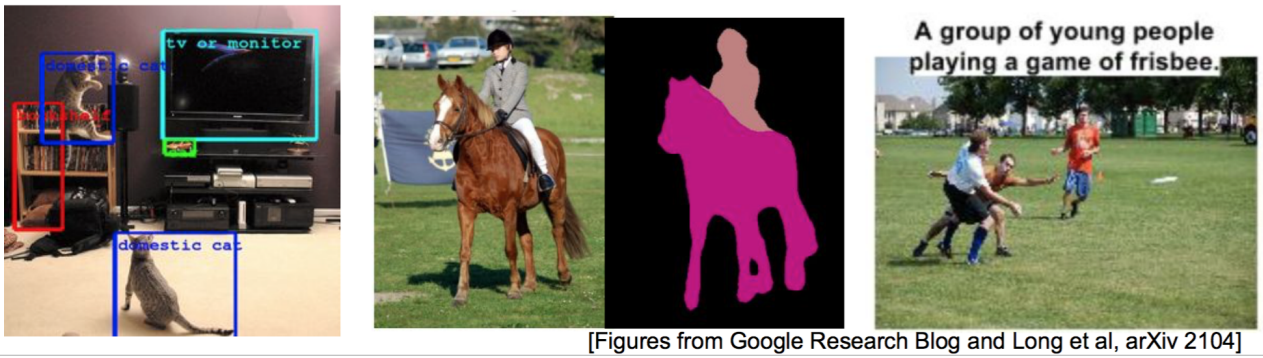
Feature Generalization: Task

- There is a large-scale dataset (ImageNet) for classification, but there are only small-scale datasets for other tasks (e.g., detection, segmentation).
- Pre-train a CNN on a large-scale classification dataset.
- Use the pre-trained CNN as "feature extraction" method for other tasks.



Feature Generalization: Task

- Achieves state-of-the-art results on many other vision tasks.
  - ex) object detection, segmentation, depth map prediction, image caption generation
- A CNN trained on a large-scale classification dataset learns a generic feature that can be used for many different vision tasks.



[Figures from Google Research Blog and Long et al, arXiv 2104]

Summary of Convolutional Neural Network

- Convolutional Neural Network: a special kind of neural network with local connectivity and weight sharing.
- Achieves state-of-the-art performances on many different computer visoin tasks.
- Higher layers extract high-level features (e.g., dog face).
- Learned features can be generally used for other vision tasks.